SCore and PM Programming Manual
– DRAFT –

Atsushi HORI

November, 2006

Contents

List of Figures

List of Tables

Preface

SCore (pronounced as [es-core]) was designed to be an operating system for clusters for high performance computation from the beginning. Thus SCore software philosophy and architecture are unique and very different from the other cluster management software or parallel programming environment for clusters.

This document was written for readers those who will develop runtime systems or parallel programming (runtime) environments.

Here is the assumption of the readers of this document.

- Having enough knowledge of Linux, C programming language and parallel programming
- Having the experience of using SCore cluster
- Trying to write a program or a runtime library to use PMv2 to get higher efficiency

Note

The sample programs in this documents are tested with SCore 6.0.1. The earlier versions of SCore may not work properly.

Chapter 1

Introduction

1.1 SCore Overview

SCore is a cluster operating system software package for high-performance clusters. SCore was originally developed in a Real World Computing Project funded by Japanese government, and now SCore is being developed and maintained by PC Cluster Consortium (`http:www.pccluster.org`). SCore was designed and is being developed to be an all-in-one package which supports almost everything needed for cluster computing. However, SCore does not provide cluster management functions which depends on hardware too much.

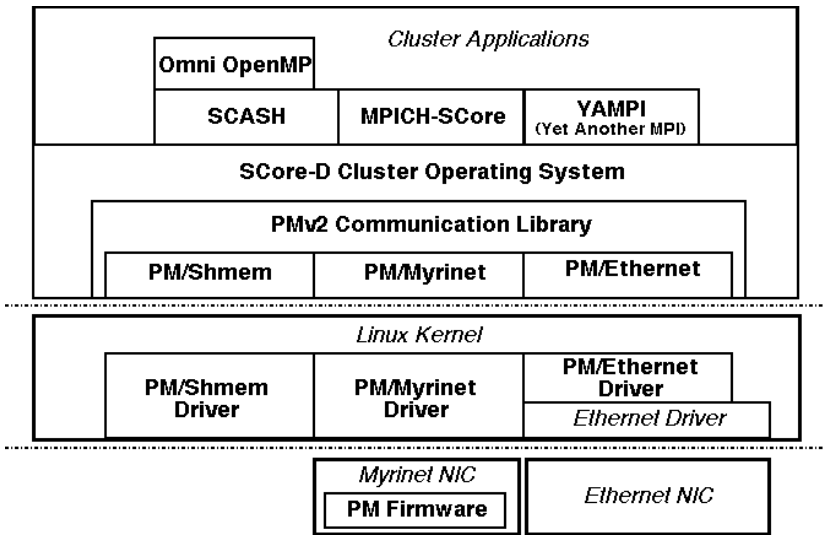

Figure 1.1: SCore Software Architecture

Figure 1.1 shows the software architecture of SCore. The most popular parallel programming environments, MPI and OpenMP are included in the SCore package. There are two MPI implementations on SCore, MPICH-SCore based on MPICH (`http://www.mcs.anl.gov/mpi/mpich2`), and YAMPI, `http://www.il.is.s.u-tokyo.ac.jp/yampii/`). OpenMP is a parallel programming environment for shared memory machines. However, Omni

OpenMP compiler (`http://phase.hpcc.jp/Omni/home.html`) enables OpenMP programs to run on distributed memory clusters.

SCore-D is a cluster operating system running on top of Linux operating system. SCore-D manages various cluster resources and schedules parallel jobs. SCore jobs can be submitted by the third-party batch job schedulers, such as PBS(Pro), NQS, LSF, SGE, etc.

SCore-D also supports checkpoint/restart and migration. Further, SCore-D supports network and processor heterogeneity. For example, cluster may have the mixture of Myrinet and Ethernet, and/or the mixture of Intel/Pentium processors and AMD/Opteron processors.

PMv2 is a low-level, high-performance communication library in SCore. It supports various network devices, such as Myrinet and Ethernet.

Note on SCore

- SCore is a cluster system software package
- SCore is not a cluster *management* software package
- SCore supports MPI and OpenMP parallel programming environments
- SCore supports checkpoint/restart and migration
- SCore has its own job scheduling
- SCore can operate with the combination of third-party batch job schedulers
- SCore supports network and processor heterogeneities

1.2 PMv2

PMv2 is the name of API for high-performance, low-level communication library. It supports multiple (physical layer) protocols as shown in Table 1.1. This allows users to have only one binary executable file which can run on clusters having different network device. The PMv2 communication is neither peer-to-peer nor connection oriented. PMv2 communication preserves message order.

1.2.1 PM Device

There are various PM devices, Ethernet, Myrinet, Infiniband, and Shmem. PMv2 allows to design a PM device to be implemented as a user-level communication or kernel-level communication. Indeed, PM/Myrinet is implemented as a user-level communication, while PM/Ethernet is a kernel-level communication.

1.2.2 PM Context

There are two kinds of PM objects, device and context. PM device can be thought as a *class object* in an object-oriented language and PM context is an *instance object* derived

Table 1.1: PMv2 Devices

Device Name	Physical Device	Comment
shmem	Shared Memory	Intra-host communication
myrinet	Myrinet(2K,XP,2XP)[1)]	
ethernet	(10,100,Gb,10G) Ethernet	
ethernet-hxb	(10,100,Gb,10G) Ethernet	Experimental
ib-ts	Top Spin[2)] (Mellanox[3)]) Infiniband	Experimental
infiniband-fj	Fujitsu Infiniband	Experimental
sci	SCI	Experimental
agent-udp	Ethernet (UDP/IP)	Experimental

[1)] `http://www.myri.com/`
[2)] `http://www.topspin.com/`
[3)] `http://www.mellanox.com/`

from a class object. PM context can also be thought as an *end-point* in terms of network communication and actual communication operations are implemented as methods of PM context.

1.2.3 PM Functions

From the beginning, PMv2 was designed assuming the existence of SCore-D, a cluster operating system. There are 61 functions defined in PMv2, however, almost half of them are dedicated for the use of SCore-D (Table 1.2). PMv2 is designed not only for high-performance, cluster in mind communication library, but also supporting system-level checkpoint/restart, migration and gang-scheduling. The object of this document is to explain how to use the PMv2 communication library to develop a runtime library or a user program using PMv2 directly. The functions to implement checkpoint/restart, migration and gang-scheduling are designed to be used by the SCore-D runtime library. Thus describing the PM functions for SCore-D are thought to be out of the scope of this document and these should be described in the PMv2 device development document.

Most PM functions return `PM_SUCCESS` if they succeed and return `ENOSYS` if the function is not available on the PM object (PM device or context).

1.2.4 PM and SCore-D

In modern operating systems, such as Unix and Linux, interaction of different processes are very restricted and it should be done via various inter-process communication methods. SCore-D and PM were co-designed so that a cluster operating system should have the same integrity as modern operating system for sequential machines. Thus PM devices are opened by SCore-D and PM contexts are created by SCore-D and passed to user processes for the communication between the processes. Any PM contexts passed to a user process are shared with SCore-D so that SCore-D can investigate the context and save the communication status into memory or disk when the parallel job is checkpointed or gang-scheduled.

Table 1.2: PM Functions

	Function Name	Privilege	Function Name	Privilege
Device Ops.	`pmGetTypeList()`	SCore-D	`pmCloseDevice()`	SCore-D
	`pmGetDeviceConfig()`	SCore-D	`pmGetNodeList()`	SCore-D
	`pmGetOptionBit()`	SCore-D	`pmIsReachable()`	SCore-D
	`pmOpenDevice()`	SCore-D		
Context Ops.	`pmAddNode()`†	User	`pmAttachContext()`	User
	`pmBindChannel()`	SCore-D	`pmCloseAttachFd()`	SCore-D
	`pmCloseContext()`	SCore-D	`pmControlReceive()`	User
	`pmControlSend()`	SCore-D	`pmCreateAttachFd()`	SCore-D
	`pmDetachContext()`	User	`pmExtractNode()`†	User
	`pmGetContextConfig()`	User	`pmGetMtu()`†	User
	`pmOpenContext()`	SCore-D	`pmIsSendStable()`	SCore-D
	`pmRemoveNode()`†	User	`pmResetContext()`	User
	`pmRestoreContext()`	SCore-D	`pmSaveContext()`	SCore-D
	`pmUnbindChannel()`	SCore-D		
Message Passing	`pmAfterSelect()`	User	`pmAssociateNodes()`	User
	`pmBeforeSelect()`	User	`pmGetFd()`	User
	`pmGetMessageQueueStatus()`	User	`pmGetSelf()`	User
	`pmGetSendBuffer()`	User	`pmReceive()`	User
	`pmReleaseReceiveBuffer()`	User	`pmTruncateBuffer()`	User
	`pmIsSendDone()`	User	`pmSend()`	User
Remote Memory Access	`pmIsReadDone()`‡	User	`pmIsWriteDone()`‡	User
	`pmMLock()`‡	User	`pmMUnlock()`‡	User
	`pmRead()`‡	User	`pmWrite()`‡	User
Checkpoint/ Restart and Migration	`pmCheckpoint()`‡	User	`pmGetMmapInfo()`‡	User
	`pmMigrateSys()`‡	SCore-D	`pmMigrateUser()`‡	User
	`pmRestartSys()`‡	SCore-D	`pmRestartUser()`‡	User
Debug	`pmDebug()`	User	`pmDumpContext()`	User
Misc.	`pmErrorString()`	User		

†PM/Composite only
‡Optional Function

1.2.5 PM/Composite Device

There is another PM device not shown in Table 1.1, called PM/Composite which plays very important role in SCore, however, in most cases it is invisible from users. PM/Composite device is also called "pseudo device" since it has no physical network device and no communication ability. Instead, the context of PM/Composite has a routing table and can hold several actual PM contexts (Figure 1.2). These PM contexts are switched according to the destination node of message sending. PM/Composite allows a program to communicate in a coherent way on SMP clusters where both inter-host and intra-host communication must take place. On an SMP cluster, SCore-D allocates a PM/Composite context, PM/Shmem contexts and other PM contexts. And the initializing function of SCore runtime library receives the PM contexts from SCore-D and compose the PM/Composite context with other PM contexts which can communicate withe other nodes.

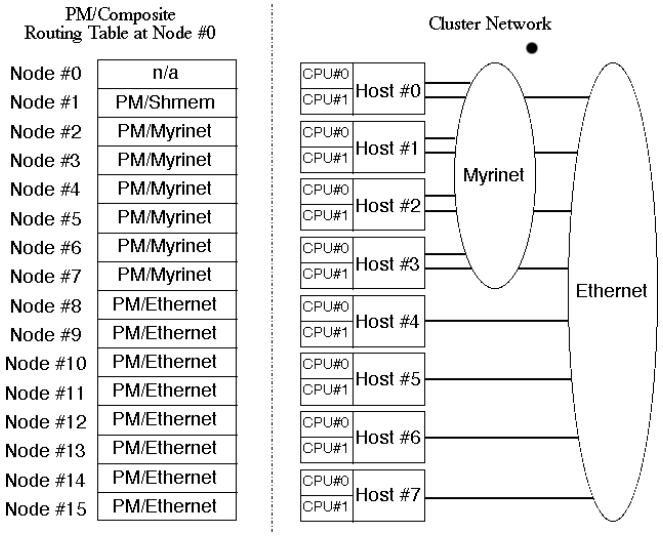

Figure 1.2: Example of PM/Composite

1.2.6 Network Heterogeneity

It is the natural extension of the idea for PM/Composite to support multiple PM devices, such as PM/Myrinet and PM/Ethernet. There could be a case where a faster PM network covers part of a cluster and slow network covers all the cluster, because of small budget which is not enough to buy faster but expensive network covering whole cluster, for example. In SCore, the two networks can be utilized by using PM/Composite so that Myrinet which is much faster than gigabit Ethernet is used as much as possible.

Figure 1.2 shows an example of how PM/Composite context is configured with having multiple PM networks, PM/Shmem, PM/Myrinet, and PM/Ethernet. Here, cluster has two physical networks, Myrinet and Ethernet, but only half of the cluster is covered by Myrinet (right hand side of Figure 1.2. In the SCore initialization, the routing table of PM/Composite context on the zeroth node is set up to utilize both networks as shown in the left hand side of Figure 1.2.

Note on PMv2 Protocol

- Connection Less Protocol
- Preserving Message Order
- Supporting Zero-Copy Communication
- Supporting Multiple Protocols
- Supporting Checkpoint/Restart, Migration and Gang-Scheduling
- Supporting Network Heterogeneity

1.3 Heterogeneity

SCore supports not only the heterogeneity of network, but also supports processor heterogeneity. PMv2 and SCore-D are designed so that an SCore cluster can consist of the mixture of CPUs with different architectures. SCore can handle the differences of word length (32 bits or 64 bits), endian (big or small), and OS (different distributions, different versions).

If a runtime library or application which uses PM directly should be carefully implemented with those processor heterogeneity, if the library or application supports the heterogeneity, especially for word length and endian.

`smake` command which is a wrapper script (named `.wrapper`) of the Linux `make` command supports to produce different binaries for each different architecture and/or OS (Man page:49).

1.4 Program Invocation Procedure

1.4.1 Programmed Resource Request

Some parallel programs have the constraint on the number of nodes required to run. For example, in NAS parallel benchmark suite (`http://www.nas.nasa.gov/Resources/Software/npb.html`), the number of nodes required to run the FT (Fourier Transform) program must be the power of 2, and some others require square numbers. Those programs are programmed to raise an error when the number of nodes allocated is not the right one. However, there can be the case where a cluster is heavily loaded and submitted job must wait hours before it starts execution, and a user may submit a job with a wrong number of nodes. The execution terminates immediately when it starts execution because of the wrong parameter, and the user wastes the waiting time.

1.4.2 Single-User Mode

1. A user tries to submit a job by invoking `scrun` program.
2. The `scrun` process `fork()`s and `exec()`s the user parallel program on its local host to get programmed resource information.
3. The process of the user program calls `score_initialize()` function and the function collects resource request information which might be programmed in the user program.
4. The `scrun` process invokes SCore-D.
5. When SCore-D boots up, it connects with The `scrun` process and it sends resource request to SCore-D. SCore-D checks the resource request, and allocates the resources according to the request. If there is not enough resources, the job submission is failed.
6. SCore-D then `fork()`s and `exec()`s user processes on the allocated hosts. Here, SCore-D creates PM contexts for the user parallel job and passes them to the job.
7. When all the above procedure succeeds, then SCore-D starts scheduling of the job.

8. When all of the user processes are terminated, then SCore-D terminates.

1.4.3 Multi-User Mode

1. SCore-D is already running and waiting for job submission.
2. A user tries to submit a job by invoking `scrun` program.
3. The `scrun` process `fork()`s and `exec()`s the user parallel program on its local host to get programmed resource information.
4. The process of the user program calls `score_initialize()` function and the function collects resource request information which might be programmed in the user program.
5. The `scrun` process connects with SCore-D and passes resource request. SCore-D checks the resource request, and allocates the resources according to the request. If there is not enough resources, the job submission is failed.
6. SCore-D then `fork()`s and `exec()`s user processes on the allocated hosts. Here, SCore-D creates PM contexts for the user parallel job and passes them to the job.
7. When all the above procedure succeeds, then SCore-D starts scheduling of the job.
8. When all of the user processes are terminated, then SCore-Dreclaims the resources allocated to the job.
9. SCore-D is waiting for job submission.

1.4.4 Local Execution Mode

The executable binary file which is linked with SCore library can run on a host when it is invoked solely without the `scrun`. In this case, most of the SCore-D systemcall functions described in the next section do not work or simply ignored.

1.5 SCore Runtime Library Functions

The SCore-D cluster operating systems and user process has a shared memory region, called C-Area where 'C' stand for the communication between the user process and SCore-D process. In this section, the SCore runtime library which is designed to be used for the runtime-library over the SCore runtime library. The SCore runtime library is designed to be used by the upper-level runtime libraries. The functions of SCoreruntime library are listed in Table 1.3.

SCore-D is a cluster operating system. SCore-D not only schedules user parallel jobs, but also it provides some services for user programs just like the systemcall functions in Linux. The communication between SCore-D and user process is done via C-Area. The SCore-D systemcall functions are designed to be used by the upper-level libraries and user programs. The SCore-D systemcalls are listed in Table 1.4.

Table 1.3: SCore Functions for Runtime Library

Category	Function Name	Comment
Setup	`score_initialize()`	Initialization
Idle Flag Setting	`score_become_busy()`	Setting `idle_flag`
	`score_become_idle()`	Setting `idle_flag`
Options	`score_get_opt()`	Getting `scrun` option

Table 1.4: SCore-D Systemcall Functions

Systemcall Name	Comment
`sc_barrier()`	Barrier synchronization
`sc_checkpoint()`	Trigger checkpoint
`sc_exit()`	Terminating parallel process
`sc_flush()`	Flushing `STDOUT` and `STDERR`
`sc_getpid()`	Getting job ID of SCore-D
`sc_inspectme()`	Attaching debugger
`sc_set_monitor()`	Real-time monitoring
`sc_signal_bcast()`	Broadcasting signal
`sc_sleep()`	Sleeping
`sc_yield()`	Yielding (pausing)
`sc_create_temporary_file()`	Creating a temporary file
`sc_open_temporary_file()`	Opening a temporary file
`sc_unlink_temporary_file()`	Deleting a temporary file

1.6 Parallel Process and Parallel Job

1.6.1 Standard Input

The standard input of the `scrun` process is forwarded by SCore-D to the standard input of the first node (rank 0 in MPI). The standard output of each Linux process is merged and forwarded to the standard output of the `scrun` process. The standard error output is handled in the same way as the standard output.

Some parallel programs may require to the input from `STDIN` on all processes. This case can be handled by using the very unique feature of SCore described in the next sections.

1.6.2 Parallel Process and Parallel Job

In SCore, parallel process is defined as a set of Linux processes which is derived from a parallel program. And parallel job is defined as a set of parallel processes.

```
$ scrun scatter == a.out < file.in
```

Figure 1.3: SCore pipe example

1.6.3 Redirection of Standard Input and Output

Most of the shell programs in Linux (Unix) support "I/O redirection." The same idea is also implemented in SCore. Table 1.5 shows the I/O redirection features in Linux and SCore. Almost the same functions are supported in SCore, however, the syntax is different so that the Linux shell programs can distinguish them easily.

Table 1.5: Redirection of Standard Input and Output

Linux/Unix	SCore	Comment
`a.out|b.out`	`a.out == b.out`	`a.out` and `b.out` are piped
`a.out<infile`	`a.out := file.in` [1]	`STDIN` of `a.out` is `file.in`
`a.out>outfile`	`a.out =: file.out`	`STDOUT` of `a.out` is `outfile`
`a.out>>outfile`	`a.out =:: file.out`	`STDOUT` of `a.out` is appended to `file.out`
`a.out:b.out`	`a.out :: b.out`	Sequential invocation of `a.out` and `b.out`.

parallel job in defined in SCore is piped and/or serialized parallel processes and the parallel job is the unit of SCore-D scheduling. All parallel processes in a parallel job are running on the same node allocated by SCore-D. When two parallel processes are connected by an SCorepipe, then the processes running on the same node are connected with the Linux pipe.

1.6.4 Built-in `scatter` Program

As described in the beginning of this subsection, SCore supports `STDIN` forwarded to the `STDIN` of the first node of a parallel process. However, combination of the parallel job described in the previous subsection and the `scatter` program (Man page:83) which is built in the SCore package, the `STDIN` can be forwarded to the `STDIN` of all processes in a parallel process. The example of the `scrun` command invocation of a piped parallel job is shown in Figure 1.3.

The `STDIN` of `scrun`, which is redirected to the file named `file.in` located on the host where the `scrun` is invoked, is forwarded to the `STDIN` of the first node of `scatter` parallel process, and the `scatter` program broadcasts its `STDIN` to the `STDOUT` of all the processes of the parallel process. The `STDOUT` of the `scatter` program on each node is connected with the Linux pipe with the `STDIN` of the `a.out` on the same node. Thus the `file.in` file becomes the `STDIN` of every process of the `a.out` parallel process.

In this example shown in Figure 1.4, each `STDOUT` of the `scatter` parallel process is redirected to the `file.out` file which is located on the every host where the parallel process runs. The effect of this example is very similar to the Linux `rdist` program.

```
$ scrun scatter =: file.out < file.in
```

Figure 1.4: SCore redirection example

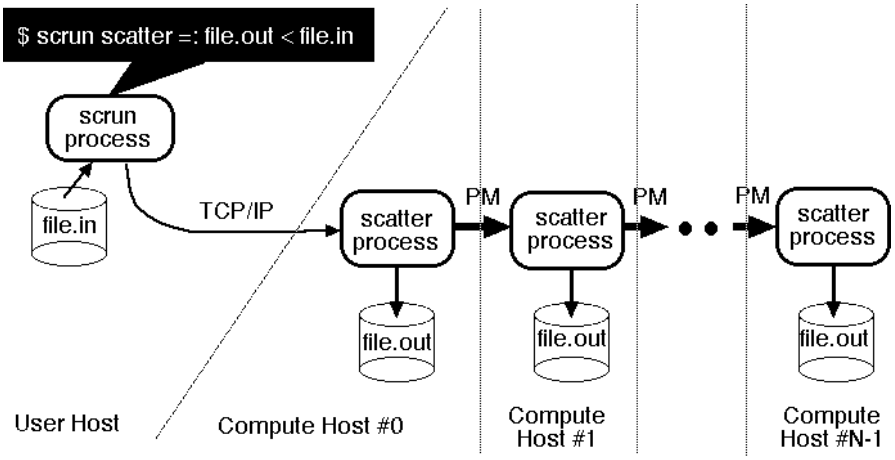

Figure 1.5: Communicating Processes in `scatter` command

The Linux processes in this example and data flow between them are shown in Figure 1.5. The file content is copied in a pipe-lined way. The `scrun` program forwards its `STDIN` via a TCP/IP connection to the first node running on a compute host. The first node reads the file content from its `STDIN`, writes to a local file and then it passes the content to the next node via PM communication. The next node writes to a local file and then forwards the content to the next node. This forwarding is repeated in parallel until it gets to the final node.

From the next section, this `scatter` is focused as examples of SCore and PMv2 programming. This program is simple and considered to be a good working example of SCore and PMv2 programming. The actual `scatter` program is a little bit more complicated though.

Chapter 2

Quick Programming

2.1 Initialization

```
#include <score.h>
void score_initialize( void )
```

Figure 2.1: `score_initialize()`

Every SCore program must call `score_initialize()` function at the very beginning of the program. `score_initialize()` has two different rolls depending of the context how the program is invoked.

- When the user program is invoked by the `scrun` command and runs local (user) host, then the SCore runtime library collects programmed resource request and pass the resource information to the `scrun` process. Then the user program terminates in the SCore library.
- When the user program is invoked by SCore-D on compute hosts, then it get PM contexts allocated by SCore-D.

The obtained resource information is passed to SCore-D when the `scrun` tries to submit the job. SCore-D then check the resource request, if there are enough resources then the job is accepted. User program, however, need not to distinguish those two cases, because `score_initialize()` function never returns in the case of local execution to get resource information.

```
#include <score.h>

int main( int argc, char **argv ) {
  score_initialize();
  ....
}
```

Figure 2.2: SCore Initialization

After returning from the `score_initialize()` function, although this means the program is executed on compute hosts, some global variables are set (Table 2.1). Allocated PM contexts are set to an array variable named `score_pmnet`, and are ready to communicate with any other nodes and no need of extra initialization.

Each element of the `score_pmnet` array variable always points to a PM/Composite contexts and the PM/Composite contexts hold actual PM contexts. The PM/Composite contexts are set inside the `score_initialize()` function so that the process can communicate via the PM/Composite context with every allocated hosts and created processes by SCore-D. However, trying to communicate with a node which is not allocated by SCore-D will result in an error.

A PM/Composite context and the PM contexts held by the PM/Composite context are called network set. A program may have several network sets. For example, a program can have two network sets, one of them is used for barrier synchronization and another is used for massive data transfer.

Table 2.1: Global variables defined in `score.h`

Variable Name	Comment
int score_self_node	node number of self process
int score_self_proc	process number on host
int score_self_host	host number
int score_num_node	number of nodes allocated
int score_num_proc	number of processes on host
int score_num_host	number of host allocated
int score_num_pmnet	number of PM network sets allocated
pmContext *score_pmnet[]	PM contexts

The `score_num_pmnet` is set to zero and `score_pmnet[0]` is set to `NULL` when only one node (one process) is allocated by SCore-D. Since there is no way to communicate with only on node and no PM context is allocated.

2.2 Message Passing

2.2.1 Naive Message Receiving

There are two functions to receive a PM message, `pmReceive()` and `pmReleaseReceiveBuffer()`. The `pmReceive()` function tries to receive a message. If there is one or more received messages in the receive buffer, the function returns `PM_SUCCESS`. Unless it returns `ENOBUFS` or `EBUSY` (see also Section 2.3.1). The returned buffer address is word-aligned and able to cast in any type. The received message must be released by calling the `pmReleaseReceiveBuffer()` function. The `pmReceive()` and `pmReleaseReceiveBuffer()` functions may return some other error numbers. Unfortunately those are up to PM devices and not well-defined.

In Figure 2.4, a very simple and naive function `naive_recv_message` is defined to receive a PM message. As in the previous example, the function shown here is to give readers an intuitive realization, and no good for real use in the sense of healthiness and efficiency. A proper communication code will be shown later in this chapter.

```
#include <score.h>
int pmReceive( pmContext *pmc, caddr_t *bufp, size_t *lenp );
int pmReleaseReceiveBuffer( pmContext *pmc );
```

Figure 2.3: `pmReceive()` and `pmReleaseReceiveBuffer()`

```
#include <string.h>
#include <errno.h>
#include <score.h>

int naive_recv_message( int dest, void *message, size_t *len ) {
  caddr_t sbuf;
  int cc;

  while( 1 ) {
    cc = pmReceive( score_pmnet[0], &sbuf, &len );
    if( cc == ENOBUFS || cc == EBUSY ) {
      continue;
    } else if( cc == PM_SUCCESS ) {
      memcpy( message, sbuf, len );
      cc = pmReleaseReceiveBuffer( score_pmnet[0] );
    }
    break;
  }
  return( cc );
}
```

Figure 2.4: `naive_recv_message()`

All the information in the received message must be extracted or copied by the time of calling the `pmReleaseReceiveBuffer()` function. The buffer region for the received message is reclaimed and recycled when the `pmReleaseReceiveBuffer()` function is called.

2.2.2 Naive Message Sending

Similar to the PM message sending in the previous subsection, There are two functions to send a PM message via a PM context, `pmGetSendBuffer()` and `pmSend()`. The `pmGetSendBuffer()` function is called to get a memory space to send a message. The message buffer space is word-aligned and able to cast in any type. The `pmSend()` function is to send the message which was obtained via calling the `pmGetSendBuffer()` function. As in the PM message sending, calling of the `pmGetSendBuffer()` and `pmGetSendBuffer()` functions must be paired at any time. Any PM device does not allow to send a message to the node itself. No zero length message is allowed to send. In both cases, `pmGetSendBuffer()` function returns `EINVAL`. The `pmGetSendBuffer()` and `pmGetSendBuffer()` functions may return some other error numbers. Unfortunately those are up to PM devices and not well-defined.

```
#include <score.h>
int pmGetSendBuffer( pmContext *pmc, int node, caddr_t bufp*, size_t len )
int pmSend( pmContext *pmc )
```

Figure 2.5: `pmGetSendBuffer()` and `pmSend()`

In Figure 2.6, a very simple function `naive_send_message` is defined to send a PM message. The function shown here is to give readers an intuitive realization, and no good for real use in the sense of healthiness and efficiency. A proper communication code will be shown later in this chapter.

```
#include <string.h>
#include <errno.h>
#include <score.h>

int naive_send_message( int dest, void *message, size_t len ) {
  caddr_t sbuf;
  int cc;

  while( 1 ) {
    cc = pmGetSendBuffer( score_pmnet[0], dest, &sbuf, len );
    if( cc == ENOBUFS || cc == ENOBUFS ) {
      // processing received message must take place here !!
      continue;
    } else if( cc == PM_SUCCESS ) {
      memcpy( sbuf, message, len );
      cc = pmSend( score_pmnet[0] );
    }
    break;
  }
  return( cc );
}
```

Figure 2.6: `naive_send_message()`

The `pmGetSendBuffer()` function returns `ENOBUFS` or `EBUSY` when there is no room available for the sending message, and returns `PM_SUCCESS` when it succeeds (see also Section 2.3.1). The message sending in PMv2 is asynchronous. The successful return of the `pmSend()` function does NOT mean the sent message is already received by the PM context of the receiver process, but it means that the message sending is queued *successfully* in the PM context and the message may be sent in the future. The message must be constructed between calling the `pmGetSendBuffer()` and `pmSend()` functions. The modification of sending message after calling the `pmSend()` may not be reflected to the actual sent message. There is no way to cancel the queued sending message.

When the `pmGetSendBuffer()` returns `ENOBUFS` or `EBUSY`, in general, message receiving routine must be called to avoid a deadlock. For example, think about a peer-to-peer communication for simplicity. Each node tries to exchange massive data which is much more than the size of MTU. When a program just repeats sending messages to send the massive data, and not trying to receive any message, soon the receive buffer of a PM context becomes full because both end never tries to extract messages in the receive buffer. And then the send buffer is also fulfilled with outstanding messages. Eventually the `pmGetSendBuffer()` keeps returning `ENOBUFS` or `EBUSY` and the program falls into a deadlock. To avoid this situation, a receiving routine should take place when the `pmGetSendBuffer()` returns `ENOBUFS` or `EBUSY`. Of course, this is not the case if it is guaranteed that any pair of two nodes never trying to send messages mutually without receiving.

2.2.3 MTU: Maximum Transfer Unit

Since PM supports multiple physical-layer protocols, the MTU (Maximum Transfer Unit) is up to PM device. Further, the PM context in the `score_pmnet` variable is the context of PM/Composite, which means actual PM device in the composite device to send messages may vary depending on the destination of the messages. The actual value of MTU depends on message destination, the `pmGetMtu()` function is used to get the MTU value of a destination node.

```
#include <score.h>
int pmGetMtu( pmContext *pmc, int node, size_t *mtup );
```

Figure 2.7: `pmGetMtu()`

Alternatively, a macro `PM_MIN_MTU` which is defined as the minimum length of MTU in the various PM devices, independent from destination node, and it is guaranteed that any message which length is less than or equal to the `PM_MIN_MTU` value can be sent to any node. The value of the `PM_MIN_MTU` is constant and programming can be easier when the macro is used, however, efficiency may be sacrificed. The MTU of PM/Myrinet is more than 8,192 bytes, but the `PM_MIN_MTU` is less than 1,500 bytes.

2.2.4 PM Macros

Table 2.2: PM Macros

Macro Name	Description
`PM_MAX_NODE`	Maximum number of nodes
`PM_MIN_MTU`	Minimum MTU among all PM devices
`PM_RMA_MTU`	MTU of remote memory access
`PM_SUCCESS`	Return status indicating success

2.3 Thread Safety

2.3.1 Thread Safety in PMv2

The PMv2 communication functions are designed and implemented to be thread-safe. There are two independent locks, one for message sending and another for message receiving, so that message sending and receiving can be overlapped. In message sending, a PM context is locked between the `pmGetSendBuffer()` function call and the `pmSend()` function call. Upon message receiving, a PM context is locked between the `pmReceive()` function call and the `pmReleaseReceiveBuffer()` function call. When a thread tries to lock a PM context which is already locked by another thread, then the `pmGetSendBuffer()` or `pmReceive()` function returns `EBUSY`.

2.3.2 Shared PM Context

On an SMP cluster, the PM contexts allocated to a process are shared with the other process(es) running on the same host. This is because each process sends a message by using the same (actual) PM context. The lock mechanism of the PM context described in the previous subsection is effective on those shared processes as well as the threads in a process. Thus it is very important to shorten the code length between the the `pmGetSendBuffer()` function call and the `pmSend()` function call and between the `pmReceive()` function call and the `pmReleaseReceiveBuffer()` function call.

2.3.3 SCore-D Systemcall

All the SCore-D systemcalls is designed and implemented to be thread-safe.

2.4 `scatter` Sample Program

As in Figure 1.5, the scatter program distributes input data in a pipelined way. First of all, a header file is shown in Figure 2.8 in which several `cpp` macros are defined.

```
#define NEXT_NODE       (score_self_node+1)
#define PREV_NODE       (score_self_node-1)
#define IS_PIPE_FIRST   (score_self_node==0)
#define IS_PIPE_MIDDLE  (score_self_node>0&&NEXT_NODE<score_num_node)
#define IS_PIPE_LAST    (NEXT_NODE==score_num_node)
```

Figure 2.8: `scatter` header file

2.4.1 `scatter`: Pipeline Version

The functions shown in Figure 2.4 and Figure 2.6 are too naive to be used in a practical application. The functions in Figure 2.9 and Figure 2.10 are more practical version of the receive and send functions. The `spinwait_receive()` is spin-waiting until it receives a message. The `score_become_idle()` function is called when there is no message available. As soon as a message is received, then the `score_become_busy()` function is called. This function must be called before calling the `pmReleaseReceiveBuffer()` function, otherwise the parallel job will be thought to be deadlocked.

The `get_send_buffer()` is also spin-waiting until it can find a room in the send buffer. Here, if the `pmGetSendBuffer()` returns `ENOBUFS` or `EBUSY`, then the function calls message receiving functions to avoid a possible deadlock. If a message can be found to receive and the `recv_func` argument is not `NULL`, then the received message is passed to the function pointed by the VARrecv_func variable to process the message.

To implement the `scatter` program of the first node, `STDIN` is read and write to a local file, and then pass to the next node. Figure 2.11 shows the sample code of doing this.

The `pmTruncateBuffer()` function is to shrink the buffer region obtained by the `pmGetSendBuffer()` function. It is not allowed to expand the buffer region. In Figure

```
#include <string.h>
#include <errno.h>
#include <score.h>

int spinwait_receive( pmContext *pmc, caddr_t *buffp, size_t *lenp ) {
  int cc;

  if( ( cc = pmReceive( pmc, buffp, lenp ) ) == ENOBUFS ||
      cc == EBUSY ) {
    score_become_idle();
    while( ( cc = pmReceive( pmc, buffp, lenp ) ) == ENOBUFS ||
           cc == EBUSY );
  }
  score_become_busy();
  return( cc );
}
```

Figure 2.9: `spinwait_receive()`

2.11 the `pmTruncateBuffer()` function is used. The `read_file_and_pass_next()` function reads entire file from the given file descriptor as its argument, and the file content is sent to the next node. In the `do` loop, firstly it allocates a send buffer region by calling `pmGetSendBuffer()` and then `read()` is called to read the file. The send buffer region is passed to the `read()` function so that memory copying is avoided. Then the buffer region is truncated to the actual size which is returned by the `read()`. And finally the send buffer is sent by calling the `pmSend()` function.

The `mtu` is set to the `PM_MIN_MTU` all the time. If the `MTU` value is obtained by calling the `pmGetMtu()` function, however, it is not correct. Even on a cluster having only one Ethernet network device, the cluster nodes can be SMP (or multi-core). This means that the allocated PM network can be the composition of PM/Shmem and PM/Ethernet and the `MTU`s can be different. Because this program is to forward the packets to the next node, and the next node forwards the packets the next of the next node. Thus it is the easiest to have the same packet size on all nodes of the pipeline. In most cases, however, getting the `MTU` by calling the `pmGetMtu()` function is the correct way.

The very first part of each packet is the byte length read actually and casted to the `(uint16_t)`. This length value is converted to the network-byte-order so that the value can be passed properly to the next host even if the byte-orders are different on nodes.

On the nodes whose node number is greater than 0, packets must be received and write to a local file then pass the packet content to the next node. This code is shown in Figure 2.13.

2.4.2 Program Termination

SCore-D assumes that a parallel process is terminated when the all processes in the parallel process are terminated. It is inconvenient to have a function which terminates the parallel job when an error happens and program can not proceed. For this purpose, there is an SCore-D system call, `sc_exit()`, to terminate a parallel process (Figure 2.14) which not

```
int get_send_buffer( pmContext *pmc,
                     int dst,
                     caddr_t *sbuf_p,
                     size_t slen,
                     int (*recv_func)( caddr_t, size_t ) ) {
  int cc;

  while( 1 ) {
    cc = pmGetSendBuffer( pmc, dst, sbuf_p, slen );
    if( cc != ENOBUFS && cc != EBUSY ) break;
    if( recv_func != NULL ) {
      while( 1 ) {
        caddr_t rbuf;
        size_t rlen;

        if( ( cc = pmReceive( pmc, &rbuf, &rlen ) ) != PM_SUCCESS ) {
          if( cc == ENOBUFS || cc == EBUSY ) break;
          return( cc );
        } else {
          if( ( cc = recv_func( rbuf, rlen ) ) != 0 ) return( cc );
          if( ( cc = pmReleaseReceiveBuffer( pmc ) ) != PM_SUCCESS ) {
            return( cc );
          }
        }
      }
    }
  }
  return( cc );
}
```

Figure 2.10: `get_send_message()`

only terminates the calling process but also terminates entire parallel process[1].

The argument of the `sc_exit()` function is passed to the `scrun` process and will be the exit value of the `scrun` process[2]. If a program is terminated by calling the normal `exit()` function, then the exit code of the first node will be the exit code of the `scrun` process.

Figure 2.15 shows the `main()` function of the `scatter` program. Together with the code shown in this, Figure 2.11 and Figure 2.13, the `scatter` is completed eventually. However, this is not sufficient yet to exhibit the full performance. Further, there are something more to take care for a program to run under SCore described in the following sections.

2.4.3 Compiling

The `scorecc` command is used to compile an SCore program written in C programming language. The `scorecc` is a wrapper script which adds appropriate include file paths and library archives needed for a program to run under SCore(Man page:49). Table 2.3 shows

[1]The `sc_exit()` was firstly introduced in SCore 6.0.0. In the earlier versions and in SCore 6.0.0, there is the `sc_terminate()` function which behaves just like the `sc_exit()` but having no argument to specify the exit code.

[2]when the `sc_exit()` function is called simultaneously on the different processes and the exit codes are different, then the exit code of the `scrun` will be one of the values and no way to specify which to win.

```
#include <sys/types.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <errno.h>
#include <score.h>

int read_file_and_pass_next( pmContext *pmc, int fd_in, int fd_out ) {
  size_t mtu;
  caddr_t sbuff;
  uint16_t sz;
  int cc;

  /*** this is wrong ****
  if( ( cc = pmGetMtu( pmc, next, &mtu ) ) != PM_SUCCESS ) {
    return( cc );
  }
  **** this is wrong ***/
  mtu = PM_MIN_MTU;

  do {
    cc = get_send_buffer( pmc, NEXT_NODE, &sbuff, mtu, NULL );
    if( cc != PM_SUCCESS ) return( cc );

    cc = read( fd_in, sbuff + sizeof(sz), mtu - sizeof(sz) );
    if( cc < 0 ) return( errno );       // read error
    sz = (uint16_t) cc;
    *(uint16_t*) sbuff = htons( sz );
    if( sz > 0 ) {
      cc = write( fd_out, sbuff + sizeof(sz), (size_t) sz );
      if( cc != sz ) return( errno );   // write error
    }
    cc = pmTruncateBuffer( pmc, (size_t) ( sizeof(sz) + sz ) );
    if( cc != PM_SUCCESS ) return( cc );
    cc = pmSend( pmc );
    if( cc != PM_SUCCESS ) return( cc );
  } while( sz > 0 );
  return( 0 );          // succeeded
}
```

Figure 2.11: `read_file_and_pass_next()`

```
#include <score.h>
int pmTruncateBuffer(pmContext *pmc, size_t len);
```

Figure 2.12: `pmTruncateBuffer()`

the list of wrapper scripts and corresponding programs in SCore.

Actual compilation is done by a C compiler such as `gcc`, GNU C compiler in many cases. However, the `scorecc` command supports to use the C compilers other than the `gcc`, such as Intel compiler C or PGI C compiler. The actual C compiler can be selected by the `compiler` option. The `score_compiler_list` program lists how the `compiler` option value and the actual back-end compiler are associated (Man page:84).

```
#include <sys/types.h>
#include <arpa/inet.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <score.h>

int write_file_and_pass_next( pmContext *pmc, int fd_out ) {
  caddr_t rbuff, sbuff;
  size_t len;
  uint16_t sz;
  int cc;

  do {
    cc = spinwait_receive( pmc, &rbuff, &len );
    if( cc != PM_SUCCESS ) return( cc );

    if( ( sz = ntohs( *(uint16_t*) rbuff ) ) > 0 ) {
      cc = write( fd_out, rbuff + sizeof(sz), (size_t) sz );
      if( cc != sz ) return( errno );   // write error
    }
    if( !IS_PIPE_LAST ) {
      cc = get_send_buffer( pmc, NEXT_NODE, &sbuff, len, NULL );
      if( cc != PM_SUCCESS ) return( cc );
      (void) memcpy( sbuff, rbuff, len );
      if( ( cc = pmSend( pmc ) ) != PM_SUCCESS ) return( cc );
    }
    cc = pmReleaseReceiveBuffer( pmc );
    if( cc != PM_SUCCESS ) return( cc );
  } while( sz > 0 );            // EOF
  return( 0 );          // succeeded
}
```

Figure 2.13: `spinwait_receive()` and `write_file_and_pass_next()`

```
#include <sc.h>
int sc_exit( int exno );
```

Figure 2.14: `sc_exit()`

2.4.4 Debugging

`PM_DEBUG` Environment Variable

Unfortunately, the error numbers which PM functions return are mostly device dependent and not consistent. To understand how an error which PM functions report, there is the environment variable `PM_DEBUG`. The value of the `PM_DEBUG` variable can be an integer, one, two, three, or others (Table 2.4).

The messages produced by the PM functions are output to standard error output (`STDERR`) which is output by the `scrun` process eventually. In general, the larger the value, the more the messages output.

The environment variables set on the host where the `scrun` invoked are copied to the every compute host where the parallel execution takes place. So the setting the environment variable on the local host where the `scrun` is invoked is enough to output the debug information on every compute host.

```
#include <stdio.h>
#include <score.h>
#include <sc.h>

int main( int argc, char **argv ) {
  pmContext *pmc;
  int cc;

  score_initialize();

  if( score_num_node == 1 ) {
    fprintf( stderr, Two or more nodes required to run.\n'' );
    exit( 1 );
  }
  pmc = score_pmnet[0];
  if( IS_PIPE_FIRST ) {
    cc = read_file_and_pass_next( pmc, 0, 1 );  // 0:STDIN, 1:STDOUT
  } else {
    cc = write_file_and_pass_next( pmc, 1 );            // 1:STDOUT
  }
  if( cc != 0 ) sc_exit( 1 );
  exit( 0 );
}
```

Figure 2.15: scatter `main()`

Table 2.3: Compile Commands in SCore

Script Name	Note
scorecc	C compiler
scorec++	C++ compiler
scoref77	F77 compiler
scoref90	F90 compiler†
scoreld	Loader (Linker)

† GNU does not support F90 and you have to have a commercial compiler.

Attaching Debugger

A program bug is the result of a program which will not run in the way programmer does not expect. Thus it is very hard to expect when and where a bug of a program will appear before the program runs. Further a parallel process is a set of Linux processes and knowing which process hits a buggy code is also difficult in most case.

Using a parallel debugger such as `DDT` (Distributed Debugging Tool `http://www.allinea.com/index.php?page=48`) is a good way to find program bugs. However, SCore have a feature to debug a parallel program. The `sc_inspectme()` function is an SCore-D systemcall which asks SCore-D to attach a debugger to the process itself.

The first argument of the `sc_inspectme()` is the value of the `DISPLAY` environment and the second argument is the signal number of Linux. If the `display` argument is set to `NULL` then the value of the `DISPLAY` will be took place. The second argument is used to give users a hint assuming the error produces an exception signal and can be any value.

Table 2.4: Value of the PM_DEBUG environment variable

Value	Tag	Output messages
0	-	No information will be displayed. (Default)
1	Error	Information of unrecoverable error
2	Warning	Above and information on temporary error
3	Info	Above and any information (even if succeeded)
Higher	-	More messages may be displayed depending on PM device

```
#include <sc.h>
int sc_inspectme( char *display, int signal )
```

Figure 2.16: sc_inspectme()

The sc_inspectme() function is only effective if the scrun is invoked with the debug option. In all cases, the sc_inspectme() does not return although it has the int type.

When the debug option and the proper DISPLAY value is set and X-Window server is running on the host where the DISPLAY specifies, then an xterm window is pop up in which gdb is attached to the calling process. Or when the debug option is set but the DISPLAY is not set properly, then the gdb process attaching to the calling process is invoked and the stack trace command of gdb is executed and then the gdb command terminates. If gdb is running in an xterm window then the user can inspect the buggy program by entering gdb commands in the window. If the value of the debug options is set to ddt and DDT is properly installed, then DDT[3] is invoked instead of invoking the gdb command.

If a runtime library which is designed to run under SCore has a signal handler for the SIGSEGV signal to call the sc_inspectme() function, then a debugger can be attached automatically when the program tries to access illegal memory address, and let user know where and how the erroneous situation happens. In the runtime library of MPICH-SCore for example, the signal handler which calls the sc_inspectme() function upon receiving the SIGHUP, SIGILL, SIGFPE. SIGBUS, SIGSEGV, and SIGSYS signals. The SIGHUP signal is delivered by the SCore-D, when SCore-D detects a deadlock in a parallel process (will be described in Section 3.5.2).

Sometimes programmers put sanity check code into their programs. It is a good idea to call the sc_insepctme() function when the program violates the sanity check.

[3]DDT, Distributed Debugging Tool, is a commercial software developed by Allinea Software, UK.

2.5 Summary

Note on PMv2 Message Passing

Initialization

- `score_initialize()` must be called in the very first phase of the program.
- `score_pmnet` variable holds PM contexts, however, no PM context is available when only one node is allocated.
- When `score_initialize()` returns, Allocated PM contexts are ready to communicate.

Message Sending

- `pmGetSendBuffer()` and `pmSend()` must always be paired.
- Unable to cancel message sending
- Unable to send a message to the sender itself
- Unable to send zero length message
- MTU may vary depending on the destination node
- The message pointer obtained by `pmGetSendBuffer()` becomes obsolete when `pmSend()` returns.
- In general, when `pmGetSendBuffer()` returns `ENOBUFS` or `EBUSY`, message receiving routine must take place. Otherwise a deadlock may happen.
- The buffer region obtained by `pmGetSendBuffer()` can be truncated by calling `pmTruncateBuffer()`.

Message Receiving

- `pmReceive()` and `pmReleaseReceiveBuffer()` must always be paired.
- Unable to cancel message receiving
- Unable to change the order of message receiving
- Unable to receive zero length message
- The message pointer obtained by `pmReceive()` becomes obsolete when `pmReleaseReceiveBuffer()` returns.

Chapter 3

Advanced Programming

3.1 Signals

Some Linux signals are treated by the SCore system and behave differently from the normal Linux commands. Those signals are handled in the `scrun` program and SCore-D in different ways (Table 3.1).

Table 3.1: Signals in SCore

Signal	User → `scrun`	SCore-D → User Process
`SIGHUP`	Broadcast	Deadlock detected
`SIGINT`	Broadcast	-
`SIGQUIT`	Trigger checkpoint	Trigger checkpoint
`SIGABRT`	Broadcast	-
`SIGKILL`	(unable to catch)	Terminate parallel process
`SIGUSR1`	Broadcast	-
`SIGUSR2`	Broadcast	-
`SIGTERM`	Kill parallel job	-
`SIGCONT`	Resume parallel job	Start scheduling
`SIGSTOP`	(unable to catch)	Stop scheduling
`SIGTSTP`	Suspend parallel job	-
`SIGURG`	Broadcast	-
`SIGWINCH`	Broadcast	-

In Table 3.1, the "broadcast" in the `scrun` column means the signal is forwarded and broad-casted to the processes of the parallel job. For example, when the `SIGINT` (^`C`) is sent the `scrun` process, then the signal is broad-casted to the parallel job running on compute hosts, and eventually the parallel job is terminated because of the signal if there is no signal handler of the `SIGINT` is set in the parallel program.

The `SIGHUP` is sent to the parallel process when SCore-D detects dead lock of the parallel process (See Section 3.5.2). The `SIGQUIT` signal (^\) triggers checkpoint. The `SIGTSTP` signal (^`Z`) suspend the parallel job and the `SIGCONT` signals resume the suspended parallel job. However, the `SIGSTOP` and `SIGCONT` are used by SCore-D for scheduling.

The parallel programs running on SCore should not have a signal handler for signals, should not ignore nor block the signals used by SCore-D. For example, when a user program sets a signal handler to the `SIGQUIT` and ignores the handler of the SCore library then checkpoint never happens.

3.2 Checkpoint/Restart and Migration

In SCore before the version 6, there are some restrictions for a program to have a checkpoint (and migration). The restrictions are listed in Table 3.2.

Table 3.2: Checkpoint Restrictions (Before SCore Version 6)

1	Multi-threaded (pthread) programs
2	Program linked with a dynamic library
3	Depending on the `PID` value

In SCore Version 6 or later, a new checkpoint code is introduced and those checkpoint/migration restrictions in Table 3.2 are removed. However, some restrictions still remain as shown in Table 3.3. This is because the checkpoint before SCore Version6 is implemented at the library level, and the new checkpoint is implemented at the kernel level and library level.

Table 3.3: Checkpoint Restrictions

4	Holding a hostname
5	Holding a TCP/IP connection
6	Depending on the value of `gettimeofday()`
7	Depending on the value of `time()`
8	Opening a file and rewrite some part of it
9	Truncating a file

A program running under SCore avoid those situations as long as possible. Or if avoiding is very difficult but the restricted code can be small enough then the code region can be declared "uncheckpointable" by calling the functions in Figure 3.1. If the restricted code region is surrounded by the `score_ckpt_enter_uncheckpointable()` and `score_ckpt_leave_uncheckpointable()`, and checkpoint is triggered in the middle of the region unable to have a checkpoint, then the start of checkpoint procedure is postponed until the execution goes out of the region.

```
#include <sc.h>
score_ckpt_enter_uncheckpointable(void);
score_ckpt_leave_uncheckpointable(void);
```

Figure 3.1: `score_ckpt_enter_uncheckpointable()` and `score_ckpt_leave_uncheckpointable()`

A program can trigger checkpointing by calling the `sc_checkpoint()` function, although checkpointing can be triggered periodically or by sending the `SIGQUIT` signal to

the `scrun` process. The function returns when the checkpointing is done or the program is restarted from a checkpoint.

```
#include <sc.h>
sc_checkpoint(void);
```

Figure 3.2: `sc_checkpoint()`

3.3 Blocking Receive

There is no blocking receive function in PMv2, however, PMv2 supports blocking receive. To implement blocking receive, the Linux `select()` or `poll()` function should be used to wait for incoming messages in a blocking way. Not having some dedicated block receive functions and blocking by calling `select()` (or `poll()`) function has an advantage. A blocking receiving program can wait for not only incoming messages via PM network but also some other Linux I/O events, such as TCP connections, reading pipes, etc.

```
#include <score.h>
int pmGetFd( pmContext *pmc, int *fdarray, int *nfdp );
int pmBeforeSelect( pmContext *pmc );
int pmAfterSelect( pmContext *pmc );
```

Figure 3.3: `pmGetFd()`, `pmBeforeSelect()` and `pmAfterSelect()`

In the Linux kernel, interrupt is usually used to wake up user process which is waiting and blocking at the `select()` systemcall functions. However, this means that every time a message is received, an interrupt is raised. And the kernel should handle the number of interrupts when a parallel process frequently communicates. While the user-level communication is not using any interrupts to eliminate the interrupt handling overhead in the kernel. Thus to get low overhead, interrupts should be disabled, however, to implement blocking receive, interrupt is a must have.

To tackle this problem, PMv2 has a unique feature. The `pmBeforeSelect()` function ask the PM driver to enable interrupt (if possible). The `pmAfterSelect()` function disables interrupt. When the `select()` function is sandwiched by those functions, then the blocking receive can be implemented and the interrupt overhead can be minimized.

The file descriptors passed to the `select()` function can be obtained by calling the `pmGetFd()` function. The `fdarray` variable is a pointer to an array of file descriptors to hold the returned file descriptors, and the `nfdp` is a pointer to an integer which will be set to the number of file descriptors. Whatever the initial value pointed by `nfdp` is, the value is always set to the number of file descriptors to be returned. To get the number of file descriptors only, the `nfdp` variable of the `pmGetFd()` function should be set to zero. The number of the returned file descriptors are up to PM device and all the returned file descriptors should be passed to the `select()` function.

The `scatter` described in Section 2.4 is not blocking for receiving messages and it consumes almost 100% of CPU time. This is problematic because the `scatter` can be used with the other parallel program connected with the SCore pipe as shown in Figure

1.3. The most of the CPU power should be consumed by the program which is piped with the `scatter` program. Therefore the `scatter` program should be blocked when it waits for incoming messages. Figure 3.4 is the code of the `blocking_receive()` function which can be used as the `nonblocking_receive()` function in Figure 2.13 alternatively.

```
#include <sys/types.h>
#include <sys/time.h>
#include <malloc.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <score.h>

int blocking_receive( pmContext *pmc, caddr_t *bufp, size_t *lenp ) {
  static int *fd_net = NULL;
  static int num_fds, fd_max;
  fd_set fds;
  int cc, n;

  if( fd_net == NULL ) {        // initialization
    num_fds = 0;
    (void) pmGetFd( pmc, NULL, &num_fds ); // returns ENOSPC always
    fd_net = (int*) malloc( sizeof(int) * num_fds );
    if( fd_net == NULL ) return( ENOMEM );
    if( ( cc = pmGetFd( pmc, fd_net, &num_fds ) ) != PM_SUCCESS ) {
      return( cc );
    }
    fd_max = fd_net[0];
    for( n=1; n<num_fds; n++ ) {
      fd_max = ( fd_net[n] > fd_max ) ? fd_net[n] : fd_max;
    }
    fd_max++;
  }
  if( ( cc = pmBeforeSelect( pmc ) ) != PM_SUCCESS ) return( cc );
  while( 1 ) {
    if( ( cc = pmReceive( pmc, bufp, lenp ) ) == PM_SUCCESS ) break;
    if( cc == EBUSY ) continue;
    if( cc != ENOBUFS ) return( cc );   // Error !!

    FD_ZERO( &fds );
    for( n=0; n<num_fds; n++ ) FD_SET( fd_net[n], &fds );
    (void) select( fd_max, &fds, NULL, NULL, NULL );
  }
  if( ( cc = pmAfterSelect( pmc ) ) != PM_SUCCESS ) return( cc );
  return( PM_SUCCESS );
}
```

Figure 3.4: `blocking_receive()`

It should be noted that there is a race condition between the enabling interrupt by calling the `pmBeforeSelect()` function and the arrival of messages. Further, there can be the case where the `select()` returns a positive integer, this means there are some file descriptors which are ready to receive, but there is no arrived messages indeed. This is up to the design and implementation of a PM device driver. Thus it is not wise to depend on the returned value of the `select()` function.

3.4 Barrier Synchronization and Send Completion

The `sc_barrier()` function is the barrier synchronization systemcall of SCore-D. This barrier synchronization is not faster than the one implemented at the user application level, however, this function can be used in the very early stage of a runtime library where internal communication is not fully setup yet, but there is a need of barrier synchronization. This function is not suitable for the implementation of the `MPIBarrier()` function, for example.

```
#include <sc.h>
int sc_barrier( void );
```

Figure 3.5: `sc_barrier()`

The `pmIsSendDone()` function checks if all of the outstanding messages in a PM context are received in the receive buffer of each destination node, and returns `PM_SUCCESS` if true, and returns `EBUSY` if not true.

```
#include <score.h>
int pmIsSendDone(pmContext *pmc);
```

Figure 3.6: `pmIsSendDone()`

3.5 Deadlock Detection

3.5.1 Definition of Deadlock in SCore

A communicating process has two states, executing or waiting for messages. A process in the executing state is executing or sending messages. A process in the waiting state has nothing to do but waiting. The waiting state becomes executing state when a message is received. Messages can only be sent by a process which is executing. The executing state becomes waiting state when a process is waiting for message(s) such as answer or request from the other node(s). In a parallel process, if every process is in the waiting state and there is no messages in the network, then none of the process will have the chance to become the executing state. Thus the parallel execution of this parallel process will not proceed any more. This situation is the definition of "deadlock" in SCore[1].

If the deadlocked state of a parallel process could be detected and informed to the programmer of the parallel program, it would be some help for understanding what is happening on his/her program. Scheduling of a deadlocked parallel process is wasting CPU resource. If the deadlocked parallel process would be aborted by a cluster operating system CPU then the resource could be saved and can be allocated to the other parallel processes. Thus the CPU resource of a cluster can be utilized more. Unless the deadlock detection mechanism, a deadlocked parallel process may run hours uselessly.

[1]Precisely speaking, this is not a *deadlock* because a deadlock happens when two or more entities have a resource contention. However, according to this deadlock definition of SCore, a deadlock can happen on a parallel process consisting of only one process if it tries to receive messages from the other nodes.

3.5.2 Deadlock Detection

To detect a deadlock state of a parallel process, 1) there should be an entity out of the parallel process which detects the deadlock state, 2) the entity is able to obtain the state of individual process, executing or waiting, and 3) the entity can detect if there is a message which is not yet received by the processes.

In SCore, SCore-D can be the entity out of the parallel process, and can find if there is a message which is not yet received by the process in the parallel process, because SCore-D shares PM contexts with the processes. Thus, if SCore-D can obtain the state of the processes in a parallel process, then SCore-D can detect the deadlock state.

```
#include <score.h>
int pmGetMessageQueueStatus(pmContext *pmc, pmMessageQueueStatus
*statp);

typedef struct pm_message_queue_status {
        int     receive;    /* Received messages in queue   */
        int     send;       /* Sending messages in queue    */
        int     read;       /* There are outstanding remote reads */
} pmMessageQueueStatus;
```

Figure 3.7: `pmGetMessageQueueStatus()` and `pmMessageQueueStatus` structure

The `pmGetMessageQueueStatus()` function returns the number of outstanding messages in queues of a PM context. The value of `receive` in the `pmMessageQueueStatus` is the number of received messages which is not yet extracted by calling the `pmReceive()` function. The value of `send` is the number of messages which are enqueued by calling the `pmSend()` function but the messages which *may* not yet be enqueued into the receive buffer of its destination. The value of `read` is the number of outstanding read request (a Zero-Copy operation).

At the time of gang scheduling, SCore-D flushes the messages in the network so that the context can be swapped with the context of the other parallel process. Every time parallel processes are gang-scheduled, SCore-D checks if there are some outstanding messages in the PM context and thus SCore-D can get to know the number of outstanding messages in a parallel process.

```
#include <score.h>
void score_become_idle( void );
void score_become_busy( void );
```

Figure 3.8: `score_become_idle()` and `score_become_busy()`

Unfortunately Linux or Unix does not provide a way for SCore-D to obtain the states of processes in a parallel process. Instead, the functions to let SCore-D know the current state of the process are prepared in SCore. The `score_become_idle()` function must be called by a parallel program when it has nothing to do but is waiting for incoming messages. The `score_become_busy()` function must be called when it has something to do. Indeed those functions only set a flag, named `idle_flag`, in C-Area, so that SCore-D can sample the value of the flag. Thus SCore-D can get the state of processes and can detect the deadlock state of a parallel process.

3.5.3 Real-Time Monitor

Calling the `score_become_idle()` and `score_become_busy()` functions are up to user program (or runtime library). If a program never calls those functions, then SCore-D is unable to detect a deadlock state. However, if a program properly set the flag, then SCore-D samples the value and SCore-D can display the load information of the parallel process. This is called "real-time monitor" and when the `monitor` option is set to the `scrun` program, then `scrun` displays an X-Window which displays the activity of a parallel process in real-time. Figure 3.9 shows the example of the monitor window.

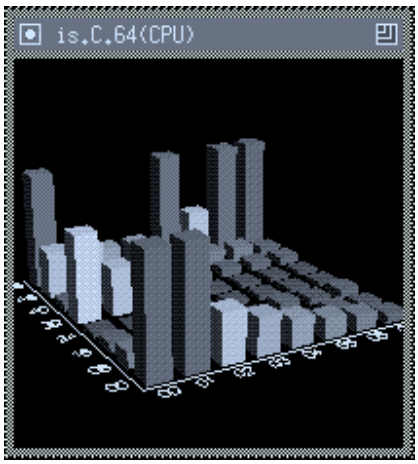

Figure 3.9: Example of Real-Time Monitor (CPU)
The higher and redder the bar, the higher the load.

In the phase of program or runtime development, the flag might be set erroneously and the execution of the program can be considered mistakenly as a deadlock. In this case, if the `noidle` option is set to the `scrun` program, and the deadlock detection mechanism is disabled.

The real-time monitoring the load of a process can be very helpful to see what is going on his/her program intuitively, because the programs using PM are usually busy-waiting for incoming messages, instead of blocking wait. And processes always consume 100% of CPU time, however, sometimes the processes are just waiting for incoming messages but has nothing to do indeed. The Linux `ps` or `top` commands are useless to see how mush user program is busy or not actually.

SCore-D can display monitor window not only for the CPU load, but also communication frequency, memory and disk usage. As shown in Table 3.4, `scrun` can accept those `monitor` option values. The option values of `usr0` and `usr1` are used to display the values which are set by a user program. The `sc_set_monitor()` function sets the value, in the range of 0 to 255, to be displayed in the real-time monitor window.

```
#include <sc.h>
void sc_set_monitor(int which, unsigned char value);
```

Figure 3.10: `sc_set_monitor()`

Table 3.4: `scrun` Real-Time Monitor Option Values

Option Value	Monitoring
load	CPU Usage
comm	Communication Frequency
memory	Memory Usage
disk	Disk Usage
usr0	The value set by user program
usr1	The value set by user program

3.6 Optional PM Operations

PMv2 allows for a PM device to have some optional functions. In this section, the usage of those optional functions are described. However, a program should not depends solely on those functions. Otherwise the program would depend on some specific PM devices.

3.6.1 Attributes of PM Context

```
#include <score.h>
int pmGetContextConfig( pmContext *pmc, pmContextConfig *configp );
```

Figure 3.11: `pmGetContextConfig()`

The `pmGetContextConfig()` function is to obtain the information of a PM context. It returns `pmContextConfig` (Figure 3.12). Here, the most important member of the structure is the `option` bits. Each bit of the variable represents the availability of an optional function.

```
typedef struct pm_context_config {
  const char *type;       /* PM device name    */
  int     number;         /* Context number    */
  int     nodes;          /* Number of nodes   */
  size_t  mtu;            /* MTU               */
  size_t  size;           /* Size of context   */
  unsigned long option;   /* Supported options */
} pmContextConfig;
```

Figure 3.12: `pmContextConfig` structure

Table 3.5 lists the macros defining the option bits and corresponding function. If the bit is set, the function is available on the context. Otherwise the calling of the function returns `ENOSYS`. In PM/Composite, the option bits are the bit-wise `AND` of the member PM contexts. Note that some of the option bits are reserved for the internal use.

Table 3.5: Option Bits

Define Macro	Function
PM_OPT_REMOTE_WRITE	Remote Memory Write (One-Sided)
PM_OPT_REMOTE_READ	Remote Memory Read (One-Sided)

```
unsigned long get_pm_option_bits( pmContext *pmc ) {
  pmContextConfig config;
  int cc = pmGetContextConfig( pmc, &config );
  if( cc != PM_SUCCESS ) return( 0 );
  return( config.option );
}
```

Figure 3.13: get_pm_option_bits()

3.6.2 One-sided Communication

The "Remote Memory Write" operation allows a program to write memory content to the memory region on a remote host. The "Remote Memory Read" operation allows a program to read memory content from a remote host. Those operation is also called "one-sided communication" which means there is no need of programming of explicit protocol handling on a remote host. The term of "zero-copy communication" is also used, however, the number of copying by software depends on the PM device, thus the term of "zero-copy" is not precise.

Table 3.6: Optional Functions

PM Device	Remote Memory Write	Remote Memory Read
Shmem	no[2]	yes
Myrinet	yes	yes
Ethernet	yes	yes

Table 3.6 shows which PM device has which optional function(s). Myrinet has a processor and DMA engines on its NIC, and it is unnecessary to copy messages by software. On the other hand, PM/Shmem has no physical device nor DMA engine, a message is copied only once from a source process to a destination process on the same host. This one-copy message transfer is done by the PM/Shmem kernel driver to be one-sided. The remote memory access of PM/Ethernet is also done by PM/Ethernet kernel driver to implement one-sided communication.

3.6.3 Ordering Rule

In PM's normal message passing, message order on the same source node and destination node is preserved. As in the normal message send receive, the order of remote memory operations is also preserved. The remote memory read operation after a remote memory write operation on the same memory region should reflect the memory content of the previous remote memory write operation. However, this is not the case when normal message sending and remote memory access operation to the same node are intermixed.

When a remote memory access operation takes place after a normal message sending, the remote memory access operation *MAY* overtake the previous normal messages. On the other hand, a normal message sending after a remote memory access never overtake the previous remote memory access.

3.6.4 PM Address Handle

All memory region which is the target of remote memory operation must be pinned down in advance. Since the pinning memory pages is a costly operation, frequent pinning and unpinning operations can result in additional overhead. To avoid this overhead, PM/ introduced pin-down cache in which pinned memory pages are placed in the cache and actual unpinning memory pages are postponed until the cache overflows. Thus pinning memory pages which is in the cache do not have to pin down.

```
#include <score.h>
int pmMLock(pmContext *pmc, int rmt_node, caddr_t addr, size_t len,
pmAddrHandle *hndlp);
int pmMUnlock(pmContext *pmc, int rmt_node, caddr_t addr, size_t len);
```

Figure 3.14: `pmMLock()` and `pmMUnlock()`

The `pmMLock()` function is to lock (pin-down) memory pages and the `pmMUnblock()` function is to unlock (unpin) memory pages. Note that the address (`addr`) and length (`len`) of the pinned memory region must be page-aligned. The `pmMLock()` function returns `pmAddrHanlde` structure which points to the locked (pinned) memory region. The `pmMLock()` function and `pmMUnlock()` function must always be paired properly.

The actual memory pinning operation is up to PM device. Since actual PM device in a PM/Composite to communicate is chosen according to the destination, the remote node (`rmt_node`) must be specified to lock or unlock a memory region. Instead, `PM_NODE_ANY` can be specified as a remote node. In this case, the lock or unlock takes place on all PM devices which a PM/Composite context has. Specifying `PM_NODE_ANY` is easier to program, however, it incurs additional overhead to lock or unlock.

The `allocate_locked_buffer()` function allocates a buffer region by calling the `posix_memalign()` function which can allocate a memory region with the specified alignment. In this case, the region is page-aligned because the entire allocated memory region can be locked. Then the memory region is locked by calling the `pmMLock()` function. In this case, `PM_NODE_ANY` is used to specify the node number.

PM address handle can be sent to the other node by the PM's message passing without taking care of the byte order. Figure 3.16 shows the `send_local_handle()` and `exchange_handle()` functions to send and receive a PM address handle.

3.6.5 One-sided Communication

The `pmWrite()` function writes the local memory content pointed by `loc_handle` to the remote memory region pointed by `rmt_handle` on node `rmt_node`. The `pmRead()` function reads the remote memory content pointed `rmt_handle` into the local memory region pointed by `loc_Handel`. In each function, every memory region must be locked by calling the `pmMLock()` function described in the previous subsection. In case to send a memory

```
#include <stdlib.h>
#include <errno.h>
#include <score.h>

#define BUFF_LEN           4096 // must be power of 2

char            *buffer;
pmAddrHandle    loc_handle;

int allocate_locked_buffer( pmContext *pmc ) {
  int cc;

  cc = posix_memalign( (void**)&buffer,
                       sysconf(_SC_PAGESIZE),
                       BUFF_LEN );
  if( cc != 0 ) return( cc );
  cc = pmMLock( pmc, PM_NODE_ANY, buffer, BUFF_LEN, &loc_handle );
  if( cc != PM_SUCCESS ) return( cc );
  return( PM_SUCCESS );
}
```

Figure 3.15: `allocate_locked_buffer()`

region which is a sub region of the locked region, the `pmAddrHandle` can be added with offset.

The `pmIsWriteDone()` function returns `PM_SUCCESS` if the local memory content is sent by the `pmWrite()` to the remote node and any modification on the local content does not affect to the content of remote memory. However, this does not mean the remote memory has already been updated by the local memory content. The `pmIsWriteDone()` function returns `EBUSY` when the remote memory write operation is not yet completed.

There is no PM functions to know when the remote memory content is updated by the `pmWrite()` function. Remember the ordering rule in Subsection 3.6.3. Sending a normal messaage to the same node using the `pmGetSendBuffer()` and `pmSend()` functions after calling the `pmWrite()` function, and when the message is received at the receiver node then the memory content on the receiver node is guaranteed to be updated.

The completion of the `pmRead()` can be checked by calling the `pmIsReadDone()` function. When the local memory content is replaced by the remote content, the `pmIsReadDone()` function returns `PM_SUCCESS`, unless it returns `EBUSY`.

3.6.6 `scatter`: One-sided Version

Figure 3.20 shows the new version of the `main()` function which is capable of switching protocol according to the PM capability. At the beginning, it checks the option bits of the allocated PM network. If the allocated PM network can handle remote memory write operation, then it uses PM's remote memory write. If the network can handle remote memory read, the it uses PM's remote memory read. Otherwise it communicates with normal message passing.

```
#include <string.h>
#include <errno.h>
#include <score.h>

pmAddrHandle    loc_handle, rmt_handle;

int send_local_handle( pmContext *pmc, int dst ) {
  caddr_t buff;
  int cc;

  cc = get_send_buffer( pmc, dst, &buff, sizeof(pmAddrHandle), NULL );
  if( cc != PM_SUCCESS ) return( cc );
  memcpy( buff, &loc_handle, sizeof(pmAddrHandle) );
  return( pmSend( pmc ) );
}

int recv_remote_handle( pmContext *pmc ) {
  caddr_t buff;
  size_t len;
  int cc;

  cc = spinwait_receive( pmc, &buff, &len );
  if( cc != PM_SUCCESS ) return( cc );
  if( len != sizeof(pmAddrHandle) ) return( EIO );
  memcpy( &rmt_handle, buff, sizeof(pmAddrHandle) );
  return( pmReleaseReceiveBuffer( pmc ) );
}
```

Figure 3.16: `send_local_handle()` and `passing_handle()`

```
#include <score.h>
int pmWrite(pmContext *pmc, int rmt_node, pmAddrHandle rmt_handle,
pmAddrHandle loc_handle, size_t len);
int pmRead(pmContext *pmc, int rmt_node, pmAddrHandle rmt_handle,
pmAddrHandle loc_handle, size_t len);
```

Figure 3.17: `pmWrite()` and `pmRead()`

Scatter: Remote Memory Write

In the `rdma_write()` function, it allocates a locked (pinned down) memory region on each node. The PM address handle which points to the locked region is then passed to the previous node so that the previous node can access the region.

Figure 3.22 shows how the pipilne using the `pmWrite()` works. The `rdma_write_head()` function (Figure 3.23 must be called at the first node of the pipeline and the `rdma_write_tail()` function must be call on the other nodes. The locked buffer region pointed by the `buffer` must be confirmed to be ready by calling the `is_write_done()` function before the `buffer` content is destroyed by reading from a file or calling the `pmWrite()` function.

Scatter: Remote Memory Read

In the `rdma_read()` function, it allocates a locked (pinned down) memory region on each node. The PM address handle which points to the locked region is then passed to the next node so that the next node can access the region to fetch the memory content.

```
#include <score.h>
int pmIsWriteDone(pmContext *pmc);
int pmIsReadDone(pmContext *pmc);
```

Figure 3.18: `pmIsWriteDone()` and `pmIsReadDone()`

```
#include <errno.h>
#include <score.h>

int is_write_done( pmContext *pmc ) {
  int cc;
  while( ( cc = pmIsWriteDone( pmc ) ) == EBUSY );
  return( cc );
}

int is_read_done( pmContext *pmc ) {
  int cc;
  while( ( cc = pmIsReadDone( pmc ) ) == EBUSY );
  return( cc );
}
```

Figure 3.19: `is_write_done()` and `is_read_done()`

Figure 3.26 shows how the pipilne using the `pmRead()` works. The `rdma_read_head()` function (Figure 3.23 must be called at the first node of the pipeline and the `rdma_read_tail()` function must be call on the other nodes. The locked buffer region pointed by the `buffer` must be confirmed to be ready by calling the `is_read_done()` function before the `buffer` content is passed to the next node.

```
#include <stdio.h>
#include <score.h>
#include <sc.h>
#include ŝcatter.h''

int main( int argc, char **argv ) {
  pmContext *pmc;
  unsigned long options;
  int cc;

  score_initialize();

  if( score_num_node == 1 ) {
    fprintf( stderr, Ṫwo or more nodes required to run.\n'' );
    exit( 1 );
  }
  pmc = score_pmnet[0];
  options = get_pm_option_bits( pmc );
  if( options & PM_OPT_REMOTE_WRITE ) {
    cc = rdma_write( pmc, 0, 1 );
  } else if( options & PM_OPT_REMOTE_READ ) {
    cc = rdma_read( pmc, 0, 1 );
  } else {
    if( IS_PIPE_FIRST ) {
      cc = read_file_and_pass_next( pmc, 0, 1 );
    } else {
      cc = write_file_and_pass_next( pmc, 1 );
    }
  }
  if( cc != 0 ) sc_exit( 1 );
  exit( 0 );
}

int get_send_buffer_uint16( pmContext *pmc, int dst, caddr_t *buff_p ) {
  int cc;
  cc = get_send_buffer( pmc, dst, buff_p, sizeof(uint16_t), NULL );
  return( cc );
}
```

Figure 3.20: scatter `main()` with RDMA

```
#include <score.h>
#include ̀scatter.h''

char    *buffer;

int rdma_write( pmContext *pmc, int fd_in, int fd_out ) {
  int cc;

  if( ( cc = allocate_locked_buffer( pmc ) ) == PM_SUCCESS ) {
    if( IS_PIPE_FIRST ) {
      if( ( cc = recv_remote_handle( pmc ) ) == PM_SUCCESS ) {
        cc = rdma_write_head( pmc, fd_in, fd_out );
      }
    } else if( IS_PIPE_MIDDLE ) {
      cc = send_local_handle( pmc, PREV_NODE );
      if( cc == PM_SUCCESS ) {
        if( ( cc = recv_remote_handle( pmc ) ) == PM_SUCCESS ) {
          cc = rdma_write_tail( pmc, fd_out );
        }
      }
    } else {    // IS_PIPE_LAST
      cc = send_local_handle( pmc, PREV_NODE );
      if( cc == PM_SUCCESS ) cc = rdma_write_tail( pmc, fd_out );
    }
  }
  return( cc );
}
```

Figure 3.21: `rdma_write()`

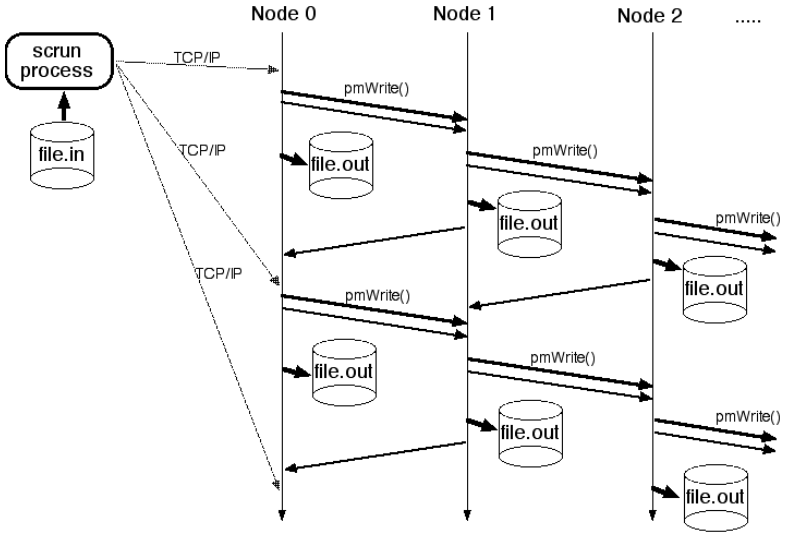

Figure 3.22: Protocol Diagram of RDMA-Write

```
#include <errno.h>
#include <score.h>
#include ŝcatter.h''

int rdma_write_head( pmContext *pmc, int fd_in, int fd_out ) {
  caddr_t buff;
  size_t len;
  int cc;

  while( 1 ) {
    cc = read( fd_in, buffer, BUFF_LEN );
    if( cc < 0 ) return( errno ); // read error
    if( ( len = (size_t) cc ) > 0 ) {
      while( 1 ) {
        cc = pmWrite( pmc, NEXT_NODE, rmt_handle, loc_handle, len );
        if( cc == PM_SUCCESS ) break;
        if( cc != ENOBUFS && cc != EBUSY ) return( cc );
      }
    }
    cc = get_send_buffer_uint16( pmc, NEXT_NODE, &buff );
    if( cc != PM_SUCCESS ) return( cc );
    *(uint16_t*) buff = htons( (uint16_t) len );
    if( ( cc = pmSend( pmc ) ) != PM_SUCCESS ) return( cc );

    if( len == 0 ) break;
    cc = write( fd_out, buffer, len );
    if( cc != len ) return( errno ); // write error

    cc = blocking_receive( pmc, &buff, &len );
    if( cc != PM_SUCCESS ) return( cc );
    cc = pmReleaseReceiveBuffer( pmc );
    if( cc != PM_SUCCESS ) return( cc );

    if( ( cc = is_write_done( pmc ) ) != PM_SUCCESS ) return( cc );
  }
  return( PM_SUCCESS );
}
```

Figure 3.23: `rdma_write_head()`

```
#include <errno.h>
#include <score.h>
#include ѐscatter.h''

int rdma_write_tail( pmContext *pmc, int fd_out ) {
  caddr_t buff;
  size_t len;
  int cc;

  while( 1 ) {
    cc = blocking_receive( pmc, &buff, &len );
    if( cc != PM_SUCCESS ) return( cc );
    len = (size_t) ntohs( *(uint16_t*) buff );
    cc = pmReleaseReceiveBuffer( pmc );
    if( cc != PM_SUCCESS ) return( cc );

    if( !IS_PIPE_LAST ) {
      if( len > 0 ) {
        while( 1 ) {
          cc = pmWrite( pmc, NEXT_NODE, rmt_handle, loc_handle, len );
          if( cc == PM_SUCCESS ) break;
          if( cc != ENOBUFS && cc != EBUSY ) return( cc );
        }
      }
      cc = get_send_buffer_uint16( pmc, NEXT_NODE, &buff );
      if( cc != PM_SUCCESS ) return( cc );
      *(uint16_t*) buff = htons( (uint16_t) len );
      if( ( cc = pmSend( pmc ) ) != PM_SUCCESS ) return( cc );
    }
    if( len == 0 ) break;
    cc = write( fd_out, buffer, len );
    if( cc != len ) return( errno ); // write error

    if( ( cc = is_write_done( pmc ) ) != PM_SUCCESS ) return( cc );

    cc = get_send_buffer( pmc, PREV_NODE, &buff, 1, NULL );
    if( cc != PM_SUCCESS ) return( cc );
    if( ( cc = pmSend( pmc ) ) != PM_SUCCESS ) return( cc );
    if( !IS_PIPE_LAST ) {
      cc = blocking_receive( pmc, &buff, &len );
      if( cc != PM_SUCCESS ) return( cc );
      cc = pmReleaseReceiveBuffer( pmc );
      if( cc != PM_SUCCESS ) return( cc );
    }
  }
  return( PM_SUCCESS );
}
```

Figure 3.24: `rdma_write_tail()`

```
#include <score.h>
#include ̀scatter.h''

int rdma_read( pmContext *pmc, int fd_in, int fd_out ) {
  int cc;

  if( ( cc = allocate_locked_buffer( pmc ) ) == PM_SUCCESS ) {
    if( IS_PIPE_FIRST ) {
      if( ( cc = send_handle( pmc, NEXT_NODE ) ) == PM_SUCCESS ) {
        cc = rdma_read_head( pmc, fd_in, fd_out );
      }
    } else if( IS_PIPE_MIDDLE ) {
      if( ( cc = send_local_handle( pmc, NEXT_NODE ) ) == PM_SUCCESS ) {
        if( ( cc = recv_remote_handle( pmc ) ) == PM_SUCCESS ) {
          cc = rdma_read_tail( pmc, fd_out );
        }
      }
    } else {
      if( ( cc = recv_remote_handle( pmc ) ) == PM_SUCCESS ) {
        cc = rdma_read_tail( pmc, fd_out );
      }
    }
  }
  return( cc );
}
```

Figure 3.25: `rdma_read()`

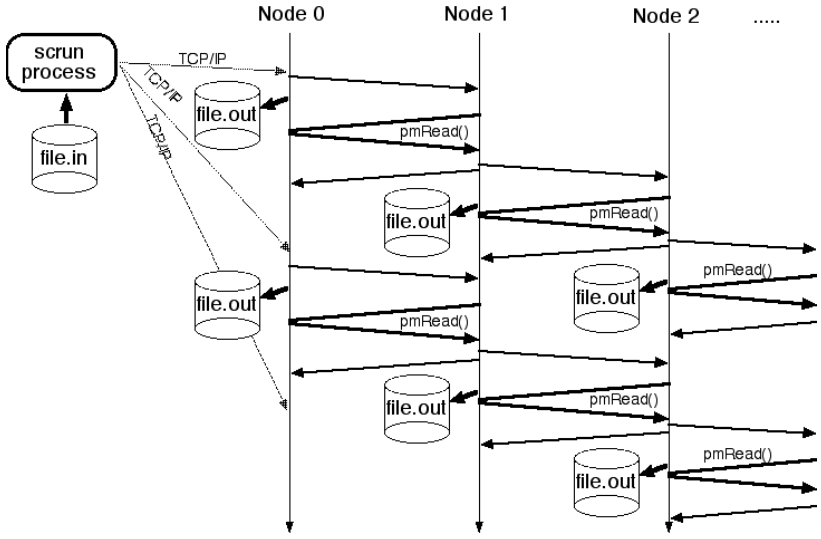

Figure 3.26: Protocol Diagram of RDMA-Read

```
#include <errno.h>
#include <score.h>
#include ѐscatter.h''

int rdma_read_head( pmContext *pmc, int fd_in, int fd_out ) {
  caddr_t buff;
  size_t len, sz;
  int cc;

  while( 1 ) {
    cc = read( fd_in, buffer, BUFF_LEN );
    if( cc < 0 ) return( errno ); // read error
    len = (size_t) cc;

    cc = get_send_buffer_uint16( pmc, NEXT_NODE, &buff );
    if( cc != PM_SUCCESS ) return( cc );
    *(uint16_t*) buff = htons( (uint16_t) len );
    if( ( cc = pmSend( pmc ) ) != PM_SUCCESS ) return( cc );

    if( len == 0 ) break;
    cc = write( fd_out, buffer, len );
    if( cc != len ) return( errno ); // write error

    cc = blocking_receive( pmc, &buff, &sz );
    if( cc != PM_SUCCESS ) return( cc );
    cc = pmReleaseReceiveBuffer( pmc );
    if( cc != PM_SUCCESS ) return( cc );
  }
  return( PM_SUCCESS );
}
```

Figure 3.27: `rdma_read_head()`

```
#include <errno.h>
#include <score.h>
#include ŝcatter.h''

int rdma_read_tail( pmContext *pmc, int fd_out ) {
  caddr_t buff;
  uint16_t l;
  size_t len, sz;
  int cc;

  while( 1 ) {
    cc = blocking_receive( pmc, &buff, &sz );
    if( cc != PM_SUCCESS ) return( cc );
    if( sz == sizeof(uint16_t) ) len = ntohs( *(uint16_t*) buff );
    cc = pmReleaseReceiveBuffer( pmc );
    if( cc != PM_SUCCESS ) return( cc );

    if( len > 0 ) {
      while( 1 ) {
        cc = pmRead( pmc, PREV_NODE, rmt_handle, loc_handle, len );
        if( cc == PM_SUCCESS ) break;
        if( cc != ENOBUFS && cc != EBUSY ) return( cc );
      }
      if( ( cc = is_read_done( pmc ) ) != PM_SUCCESS ) return( cc );
    }
    if( !IS_PIPE_LAST ) {
      cc = get_send_buffer_uint16( pmc, NEXT_NODE, &buff );
      if( cc != PM_SUCCESS ) return( cc );
      *(uint16_t*) buff = htons( (uint16_t) len );
      if( ( cc = pmSend( pmc ) ) != PM_SUCCESS ) return( cc );
    }
    if( len == 0 ) break;
    cc = get_send_buffer( pmc, PREV_NODE, &buff, 1, NULL );
    if( cc != PM_SUCCESS ) return( cc );
    if( ( cc = pmSend( pmc ) ) != PM_SUCCESS ) return( cc );

    cc = write( fd_out, buffer, len );
    if( cc != len ) return( errno ); // write error

    if( !IS_PIPE_LAST ) {
      cc = blocking_receive( pmc, &buff, &sz );
      if( cc != PM_SUCCESS ) return( cc );
      if( sz == sizeof(uint16_t) ) len = ntohs( *(uint16_t*) buff );
      cc = pmReleaseReceiveBuffer( pmc );
      if( cc != PM_SUCCESS ) return( cc );
    }
  }
  return( 0 );
}
```

Figure 3.28: `rdma_read_tail()`

Chapter 4

Other Functions

4.1 Resource Specification

As described in Section 1.4.1, resource restrictions can be specified within a user program. To do this, some resource specification macros are defined (Figure 4.1).

```
#include <score_resource.h>
SCORE_RSRC_NUM_NODES( MIN, MAX );
SCORE_RSRC_NUM_PROCS( N );
SCORE_RSRC_NUM_NETS( N );
SCORE_RSRC_ZEROCOPY;
SCORE_RSRC_HETERO_NODES;
SCORE_RSRC_NOCKPT;
```

Figure 4.1: Resource Macros

`SCORE_RSRC_NUM_NODES(`*MIN*`,`*MAX*`)` This macro limits the number of nodes to run a parallel program. The number of nodes must be greater than or equal to *MIN* value and be less than or equal to *MAX* value.

`SCORE_RSRC_NUM_PROCS(`*N*`)` This macro specifies the number of processes in a host to be allocated by SCore-D.

`SCORE_RSRC_NUM_NETS(`*N*`)` This macro specifies the number of network sets to be allocated by SCore-D.

`SCORE_RSRC_ZEROCOPY` If this is declared, then the network having the one-sided communication functions must be allocated by SCore-D.

`SCORE_RSRC_HETERO_NODES` If this macro is declared in a program, then the program can run on a heterogeneous cluster. When the heterogeneous option of the `scrun` program is specified, an error message is output.

`SCORE_RSRC_NOCKPT` If this macro is declared in a program, then checkpoint is disabled in the program. When the checkpoint option of the `scrun` program is specified, a warning message is output and the checkpoint option is ignored.

4.2 SCore-D API

Runtime Options

The `score_get_opt()` function is to get the `scrun` option. This function works just like the `getenv()` of Linux. The `keyword` argument is the keyword of the option to get, and the function returns the value if the option is present.

```
#include <score_options.h>
char *score_get_opt(char *keyword)
```

Figure 4.2: `score_get_opt()`

Flushing Standard and Error Messages

Since the `STDOUT` and `STDERR` of every process is merged and forwarded to the `scrun` process, there must be a special function to guarantee the output messages are output by the `scrun` indeed. The `sc_flush()` function does this.

```
#include <score_options.h>
char *sc_flush( void )
```

Figure 4.3: `sc_flush()`

Temporary File

When a program wants to have a temporary file whose life time is the same as the parallel job, then use the functions described in Figure 4.4. When the parallel job terminates, then the files created by the functions are deleted by SCore-D. The `filename` arguments of those functions must not contain any slash (/) character. The `sc_create_temporary_file()` creates a temporary file and the `sc_open_temporary_file()` opens an existing file and return the file descriptors. The `sc_unlink_temporary_file()` function removes the specified file.

```
#include <sc.h>
int sc_create_temporary_file( char *filename, int *fd );
int sc_open_temporary_file( char *filename, int *fd );
int sc_unlink_temporary_file( char *filename );
```

Figure 4.4: Temporary File Operation Functions

Signal Broadcast

The `sc_signal_bcast()` function broadcasts the signal specified by the `signal` argument.

```
#include <signal.h>
#include <sc.h>
int sc_signal_bcast(int signal);
```

Figure 4.5: `sc_signal_bcast()`

Sleep

The `sc_sleep()` function sleeps for the time duration specified by the `sec` argument in second. This function is similar to the `sleep()` function of Linux. However, the `sleep()` function does not work properly under the SCore-D because of the `SIGSTOP` and `SIGCONT` signals used for gang scheduling of SCore-D. Further the parallel job calling the `sc_sleep()` function will be descheduled for the specified time duration, while the `sleep()` function sleeps on the process calling the function.

```
#include <sc.h>
int sc_sleep(int sec);
```

Figure 4.6: `sc_sleep()`

Yielding

The `sc_yield()` function is similar to the `sched_yield()` function of Linux. It shall force the running parallel job to be descheduled temporarily.

```
#include <sc.h>
int sc_yield(void);
```

Figure 4.7: `sc_yield()`

Appendix A

Glossary

FEP Front End Process (FEP) communicates with SCore-D and submit a parallel job.

Host Host has a hostname and may have one or more CPUs (cores) inside.

Compute Host The hosts where a parallel program runs.

Server Host The host where several SCore server processes are running.

User Host The host where users compile their program and submit their program to run on compute hosts.

PM Context

Node Node is an entity to communicate with, in most cases, node is a Linux (Unix) process.

Parallel Job A scheduling unit which may consists of one or more parallel process.

Parallel Process A set of Linux (Unix) processes which are derived from a parallel program.

PMV2 PMv2 is a low-level communication software layer. PMv2 supports multiple protocol and has several PM devices according to the (physical level) protocol.

PM Context An endpoint of PM object consisting of send and receive buffers.

PM Device PM context is created from a PM device which encapsulate device dependent part. There are several PM devices as described below.

PM/Composite PM/Composite is a pseudo device, which means it has no actual network device but it can have several PM contexts.

PM/Ethernet PM Ethernet device.

PM/Myrinet PM Myrinet device.

PM/Shmem PM shared memory device for intra-host communication.

SCore A cluster system software package.

SCore-D A cluster operating system managing various cluster resources, hosts, CPUs (cores), PM networks. SCore-D allocates those resources to a user parallel job according to its request and schedules parallel jobs.

`scrun` A program or user command to submit SCore jobs. FEP is a process of `scrun`.

Appendix B

Man Pages

`smake(1)`

NAME

smake – make utility to maintain groups of programs for SCore

SYNOPSIS

smake [*make_options*

DESCRIPTION

The smake utility is a wrapper for the `make` command. It determines the machine's operating system type and executes make with specific SCore build options.

The options available for `smake`:

make_options command options are passed, as is, to `make`

`scorecc(1)`

NAME

scorecc – C compiler for SCore program.

SYNOPSIS

scorecc [*option* | *filename*]...

DESCRIPTION

The scorecc command compiles source files in the C language, and links object files for SCore. Almost all the options of the standard C compiler may be specified as the `scorecc` arguments.

OPTIONS

-script *script* The default setting in the *script* file is used.

-compiler *compiler*

–**compiler *compiler*** Specify the backend C compiler.

-compiler-path *compiler_command*

–**compiler-path *compiler_command*** Specify the backend compiler path. This option will override the compiler used for the compiling environment.

-scash

–**scash** Use scash library and include file.

-scash_smp

–**scash_smp** Use scash library supporting SMP cluster.

-c Compile only, no linking.

-nostatic

–**nostatic** Specify dynamic linking. `scorecc` links static by default. This option disable checkpointing facilities.

-nockpt

–**nocheckpoint** Disables all checkpointing facilities. Also disable all system call overrides for checkpoint.

-show Verbose mode. Display the commands and arguments invoked by the `scorecc` script

ENVIRONMENT

`SCORE_BUILD_COMPILERS` If the `-compiler` option is not specified, the value of this environment variable is searched. The value of `SCORE_BUILD_COMPILERS` is space or comma separated list of *script* =*compiler* or *compiler*. If *script* matches with the `-script` option value or the defasult script, then the *compiler* value is used. If there is no matched entry with *script* and only the entry of *compiler* can be found, the *compiler* value is used. Otherwise system default is used.

SCORE_COMPILERS If the `-compiler-path` option is not specified, the value of this variable is searched for. The value of SCORE_BUILD_COMPILERS is space or comma separated list of *script* =*path* or *path*. If *script* matches the -script option value or default script, the *cpath* is used. If there is no matched entry with *path* and only the entry of *path* can be found, this *path* value is used. Otherwise system default is used.

---

scrun(1)

NAME

scrun — SCore front-end process to run a user parallel program on a cluster

SYNOPSIS

scrun [-*SCoreOptions*] *program* [*program_options*]

DESCRIPTION

`scrun` is a front-end program for SCore-D that manages a variety of cluster resources. User programs running on a cluster must be invoked via the `scrun` program.

Firstly, `scrun` invokes the user program, specified by the *program* argument, on the host where `scrun` was executed. This is done in such a way to get the required resource information. Then, `scrun` tries to login to SCore-D. After login, `scrun` becomes a front-end process in order to control job status of the user program running on the cluster and output stadrad output and standard error message. When the user program finishes, `scrun` also terminates.

Valid arguments to `scrun` could be SCore options. The options are various resource specifications to SCore-D and/or options for language runtime systems on which user programs rely. In this manual page, only SCore-D options are described. The language system options must be consulted where those systems are installed.

If the first argument of `scrun` does not begin with the minus (-) character, or the third argument when SCore options are specified, then the next argument must be the filename of the program to be executed on the cluster. The specified executable file is copied and then invoked by SCore-D on all allocated nodes in the cluster. Arguments following the *program* are passed to the invocation of the executable file on all nodes.

The executable file must have read permission so that `scrun` can read the file and copy it to compute hosts. The file must also be an executable file on the host where `scrun` is invoked, so that `scrun` can execute the file to get resource information.

The `scrun` program can also submit a parallel job to compute hosts which have a different OS and/or CPU from the host where `scrun` is invoked. In this case, at least two executable files must be present, one for the `scrun` local invocation, and another for cluster execution. To allow for this situation, the executable files must be compiled with the SCore `smake`(1) commands (not `make` or `gmake`). In this case, the executable file must be a symbolic link to the `.wrapper` script which will be automatically created by the

`smake` command. It is the users responsibility to have consistent heterogeneous executable files compiled from the same source code.

GENERAL FORMAT OF SCore OPTIONS

The first character must be a minus (-), followed by keyword and value pairs, each pair is separated by a comma (,). The keyword is a predefined SCore literal and its associated value are separated by the equal (=) character. Here is an example:

```
$ scrun -nodes=2,cpulimit=4 a.out
```

In this case, two SCore options are specified, one is the "nodes" option and another the "cpulimit" option. The nodes option has the value of "2", and cpulimit has the value of "4".

If the same keywords are listed in the SCore options, then the leftmost one is taken. The value of the `SCORE_OPTIONS` environment variable is taken as the default option setting.

SINGLE USER MODE AND MULTIPLE USER MODE

The SCore-D operating system is designed to run multiple jobs at a time in a time-sharing manner. However, it has a single user mode to allow users to use the cluster exclusively. This is useful when users want to evaluate programs. When `scrun` is invoked in a `SCOUT` environment and no `scored` options are specified, then `scrun` firstly invokes SCore-D on the cluster within the `SCOUT` environment, and then the user program will be executed on the invoked SCore-D. When the user program terminates SCore-D also terminates.

If group option is sepcified, then `scrun` creates the `SCOUT` environment on the hosts specified by the value of the group option and then user program is executed in the single user mode. When the file option is specified and a set of hostnames is listed in the file specified as the option value, then the user program is executed in the single user mode in that host group. The checkpoint options is enabled when the group or file options is specified.

If SCore-D is already running on a cluster, then the user must specify, with the SCore-D option, the SCore-D server host which is accepting user logins or the hosts where SCore-D is running. User can also specify host group name by the SCore-D option to specify the set of hosts where SCore-D is running on. Precisely, the value of the `scored` option is in the format which the `scorehosts` command can accept.

RESOURCE SPECIFICATION

SCore-D manages a variety of cluster resources, such as node, networks, disks, etc. In this section, those resource specification options are described.

`nodes`[=*hosts*][**x***procs*] [**.**[*bintype*][**.** *cpugen* [**.***speed*]]]

The *hosts* is the number of hosts or nodes in a cluster required to run a user program. The *procs* is the number of processes to be invoked on a SMP cluster. If the *procs* is not present, and allocated hosts are in a SMP cluster, then the number of allocated hosts might be the number of *hosts* divided by the number of processors in the SMP

host. If the *procs* number is specified, then that number of processes on each SMP host is invoked if possible. If the number of requested nodes is less than the total number of nodes in the partition, then SCore-D allocates nodes such that load of each node is balanced. The *bintype* option specifies the binary type to be run on a heterogeneous cluster. The name of the binary type comes from the `smake` command and the `.wrapper` script. Currently, the binary type name as followed:

Proessor Type	OS	Binary Type
i386	TurboLinux	i386-turbo-linux
i386	SuSE Linux	i386-suse-linux
alpha	SuSE Linux	alpha-suse-linux
i386	Redhat Linux 7.x	i386-redhat7-linux2_4
i386	Redhat Linux 8.x	i386-redhat8-linux2_4
ia64	Redhat Linux 7.x	ia64-redhat7-linux2_4
alpha	Redhat Linux	alpha-redhat-linux
i386	NetBSD	i386-unknown-netbsd
Sparc	SunOS4	sparc-sun-sunos4
Sparc	SunOS5	sparc-sun-sunos5

On a heterogeneous cluster, users can specify CPU types by the `cpugen` option. Possible values for the *cpugen* option are specified in `scorehosts.db`, which is a database containing all cluster information. The `speed` option values are also specified in `scorehosts.db`.

`network`=*network_name*[+*network_name*]...

Users can specify the network (PM device) by the `network` option to allocate the network for user program execution on a cluster. Valid *network_name* (s) are specified in `scorehosts.db`. Users can also specify multiple networks for a user program parallel execution.

priority=*number*

Scheduling priority can be specified with the `priority` option. The smaller the value, the higher the priority. A job having a higher value will be scheduled more often.

`monitor`[=*monitor_type*]

Attach a real-time user program execution monitor. Valid types for *monitor_type* are: `load`, `comm`, `memory`, `disk`, `usr0`, `usr1`, `all` and `ALL`. The `load` value attaches a CPU activity monitor, `comm` attaches a communication activity monitor. `memory` and `disk` option values to attach memory and disk usage monitors, respectively. The usage value is scaled to limit values, if specified. Otherwise they are scaled to the values of available free space when SCore-D is invoked. The `usr0` and `usr1` options attach monitors displaying the values set by user program (See `sc_set_monitor()`). If the monitor option has no value, then load and communication monitors are attached. If user specifies `all`, then CPU, communication, memory usage and disk usage monitors are attached. If user specifies `ALL`, then all six monitors are attached. User must have an accesible X window server and the `DISPLAY` environment variable must be set correctly.

`debug`[=*number*]

The MPC++ or MPICH-SCore runtime system is programmed to detect exception signals such as `SIGSEGV`. When an exection signal is raised, the runtime system asks SCore-D to attach a `gdb` (GNU debugger) process to debug the user program. If the debug option is specified, and the user program is running in time sharing priority, then SCore-D creates a `gdb` process. Otherwise, the user program will be killed. The *number* options limits the number of debugger processes attached at the same time. The default value is 4 and the maximun number is limited to 10. If the `DISPLAY` environment variable is set, then SCore-D creates an xterm process in which `gdb` process runs. If the `DISPLAY` environment varibale is absent or having no value, but `score.gdb` file exists in the current directory and the file is readable from cluster hosts, then the `gdb` process will read the file and execute the `gdb` commands written in the file. If `score.gdb` file is not accesible, then the `gdb` process will execute only the backtrace gdb command.

`stat[istics]`[=*stat_type*]

When a user program terminates, scored outputs resource usage information to standard error or the `scrun` process. The default is to only output summary information unless *stat_type* is specified. Valid types for *stat_type* are: `all` and `detail`. If either of these types are used then individual node information will be output.

`scored`=*scored-server* [multi-user mode only]

Specify SCore-D server hostname to login SCore-D which is already running with multi-user mode. If this option is not specified, then SCore-D is invoked by `scrun` in single-user mode.

`group`=*hostgroup* [single-user mode only]

Firstly a SCOUT environment is created according to the specified *hostgroup* , then user program is invoked in the SCOUT environment. Checkpoint is enabled with this options in the single-user mode.

`restart`

Compute host sometimes crashes and running jobs are killed unexpectedly. If the restart option is set, user's program execution will be restarted from the beginning when scored is restarted with the `-restart` option. Note that this restart option is valid while the `scrun` process is alive. When the user kills the `scrun` process, restart never happens.

`checkpoint`[=*interval*]

This option is similar to the restart option, but user's program execution contexts are saved to local disk at a specified time interval. If the *interval* value is immediately followed by a character, 'm', 'h' or 'd', then the unit of the interval is minute, hour or day, respectively. When scored is restarted, program execution continues from where the more recent checkpoint was taken. This restart will only take place while the `scrun` process is alive. If you want to checkpoint in the single user mode, you must invoke `scrun` with the *group* option out of the SCOUT environment.

`cpulimit`=*limit*

Specify the CPU time *limit* (in seconds) of a user program to run.

`memorylimit`=*limit*
: This option specifies memory *limit* (in MB). This option is effective when SCore-D is running in multi-user mode.

`disklimit`=*limit*
: This option specifies disk *limit* (in MB). This option is effective when SCore-D is running in multi-user mode.

`wait`
: If the wait option is specified and login to SCore-D because of specified resource is temporarily unavailable, then login is postponed until specified resource is available. This option is only effective for SCore-D running in multi-user mode.

`message`[=*mode*]
: Control output messages produced by the SCore system at runtime. Valid modes for *mode* are: `concise` and `quiet`. The default is to output all messages. `concise` suppresses normal messages so only warning and error messages are output. `quiet` supresses all messages except for error messages.

`resource`
: When the resource option is specified, `scrun` tries to investigate SCore options and resource requests of user program(s), and then the SCore options, resource requests, and pathname(s) of user program(s) will be displayed and then exit. User program(s) will not run on a cluster.

JOB CONTROL and SIGNALS

The job status of user program execution on a cluster is linked with the job status of `scrun`. Users can suspend, resume, or kill parallel jobs running on a cluster similar to a normal UNIX command by typing ^`Z`, `fg` command, and ^`C`. Further, if the output of `scrun` is stopped by ^`S`, eventually cluster execution can be suspended until `scrun` output is allowed by ^`Q`. Typing "^\" or sending `SIGQUIT` triggers checkpointing, instead of creating a core file, and waiting for its restart when SCore-D unexpectedly terminates (system down).

Some UNIX signals delivered to the `scrun` process will be forwarded and broadcasted to the processes running in a cluster. The forwarded signals are `SIGINT`, `SIGABRT`, `SIGTERM`, `SIGURG`, `SIGWINCH`, `SIGUSR1`, and `SIGUSR2`.

INPUT/OUTPUT REDIRECTION

Similar to the Unix shell, the standard inputs and/or outputs of a parallel process can be redirected to files. When a user program specified in the `scrun` arguments is followed by the ":=" symbol and a filename, then the standard inputs of the parallel process derived from the user program are the file. If the symbol is "=:," then the standard outputs are the file. If the symbol is "=::," then the outputs are appended to the file.

Note that the open files are *local* and located in compute hosts. Further, if the filename is a basename, there is no "/" in its name, then the files are created in an SCore-D working

directory located on compute hosts, and they are removed when the parallel job is terminated. If the filename is an abosolute pathname, then the files are created on specified pathname. No relative pathname is allowed.

On an SMP cluster and the output redirection pathname is absolute, only the first process in a compute host will be redirected to the specified file, and the other processes will output to the `/dev/null`.

PARALLEL JOB

Similar to the Unix shell, `scrun` not only supports simple commands, but also pipelined commands and sequenced commands. Pipelined commands are separated with the "==" symbol, and sequenced commands with the "::" symbol. Parallel processes in a parallel job are allocated in the same partition (set of hosts) in a cluster. Pipelined parallel processes having the same node number are connected with the Unix pipe, just like the pipelined commands under the Unix shell, and they are scheduled at the same time. Parallel processes are executed in sequence when they are separated with sequential symbol(s) ("::").

Sequential programs, such as normal Unix commands can run on a cluster via the `system` comand, just like the way of the Unix `system` function. This system command can be used for house keeping of a cluster.

Combining the `scatter` command, user parallel program and `gather` command in serial, users can move necessary data file back and forth between users' workstation and compute hosts.

ENVIRONMENTS

`DISPLAY` Specify X Window server.

`SCORE_OPTIONS` Default `scrun` options can be set in this environment variable. Its value must be a list of pairs of a option name and an associated value separated by an equal (`=`) symbol. Each pair is separated by a comma (`,`). No space (blank) character must be included. The minus (`-`) symbol should NOT be placed at the begenning of the environment value.

---

sc_barrier(2)

NAME

sc_barrier — barrier synchronization

SYNOPSIS

```
#include <sc.h>
int sc_barrier( void );
```

DESCRIPTION

`sc_barrier()` must be called on all nodes to get a successful return. When the program running on all nodes reaches the point where `sc_barrier()`, then all `sc_barrier()` function calls returns.

RETURN VALUES

`sc_barrier()` returns 0 if it succeeds or returns the following error number:

ERRORS

EINTR Job is restarted from a checkpoint.

---

sc_checkpoint(2)

NAME

sc_checkpoint — trigger checkpointing

SYNOPSIS

```
#include <sc.h>
int sc_checkpoint( void );
```

DESCRIPTION

`sc_checkpoint()` triggers checkpointing.

RETURN VALUES

`sc_checkpoint()` returns 0 if the triggered checkpoint is succeeded or the program is restartde from the checkpointing.

ERRORS

---

sc_exit(2)

NAME

sc_exit — terminate the parallel job with an exit value

SYNOPSIS

```
#include <sc.h>
int sc_exit(int)
```

DESCRIPTION

The `sc_exit()` function terminates the parallel process with an exit value.

RETURN VALUES

Although the `sc_exit()` function type is `int`, it does not return.

---

sc_flush(2)

NAME

sc_flush — flush standard message and error message streams

SYNOPSIS

```
#include <sc.h>
int sc_flush( void );
```

DESCRIPTION

`sc_flush()` waits for the output of standard and error message streams on all node. Since all the message streams from compute nodes are merged and output from the scrun process, messages from different nodes is deterministic. When the `sc_flush()` function returns, then all messages previously output is guaranteed to be output.

RETURN VALUES

`sc_flush()` returns 0 if it succeeds or returns the following error number:

ERRORS

EINTR Job is restarted from a checkpoint during the flush.

---

sc_getpid(2)

NAME

sc_getpid — get a parallel process ID

SYNOPSIS

```
#include <sc.h>
int sc_getpid(int *scpid)
```

DESCRIPTION

The `sc_getpid()` function returns the parallel process ID.

RETURN VALUES

`sc_getpid()` always returns 0

ERRORS

(none)

---

sc_inspectme(2)

NAME

sc_inspectme — attach a debugger

SYNOPSIS

```
#include <sc.h>
int sc_inspectme(char *display, int signal)
```

DESCRIPTION

The `sc_inspectme()` function attaches a debugger (`GDB` or `DDT`) to the calling process, and the parallel process, including the calling process, becomes a zombie. The zombie parallel process will be destroyed when all debuggers are detached, or when the front-end process is killed. The debugger attachment takes place only when the `debug` SCore runtime option is specified, and the number of simultaneous debugger attachments is limited by the option.

When the display variable is set to `NULL`, then the value of `DISPLAY` environment is used. The signal variable is an arbitrary number to be displayed in the debugger attachment message.

RETURN VALUES

Although the `sc_inspectme()` function return type is int, it doesn't return a value.

---

sc_signal_bcast(2)

NAME

sc_signal_bcast — broadcast a signal to all processes

SYNOPSIS

```
#include <signal.h>
#include <sc.h>
int sc_signal_bcast(int signal);
```

DESCRIPTION

`sc_signal_bcast()` broadcasts signal to all processes in a user parallel process. Since SCore-D controls user processes using `SIGSTOP`, `SIGCONT` and `SIGKILL`, broadcasting one of these signals using `sc_signal_bcast()` is prohibited.

RETURN VALUES

`sc_signal_bcast()` returns 0 on success or the following error:

ERRORS

EINVAL signal is a negative number, or signal is `SIGSTOP`, `SIGCONT` or `SIGKILL`.

---

sc_sleep(2)

NAME

sc_sleep — sleep for a specified number of seconds

SYNOPSIS

```
#include <sc.h>
int sc_sleep(int sec);
```

DESCRIPTION

FUNCsc_sleep sleeps for *sec* seconds.

RETURN VALUES

`sc_sleep()` returns 0 on success or returns the following error number:

ERRORS

EINVAL *sec* is a negative number

EINTR Job is restarted from a checkpoint during the sleep.

---

sc_yield(2)

NAME

sc_yield — yield processors

SYNOPSIS

```
#include <sc.h>
int sc_yield(void);
```

DESCRIPTION

`sc_yield()` voluntarily relinquishes the processors. This results in the other runnable jobs are scheduled.

RETURN VALUES

`sc_yield()` returns 0 on success.

---

pmAddNode(3)

NAME

pmAddNode — add a node into PM/Composite context

SYNOPSIS

int pmAddNode(pmContext *pmc, int node, pmContext *member_pmc, int member_node);

ARGUMENTS

pmContext *	pmc	IN	pmContext object: must be a PM/Composite type
int	node	IN	Node number in pmc
pmContext *	member_pmc	IN	Member context
int	member_node	IN	Node number in member context

DESCRIPTION

`pmAddNode()` sets *member_node* in *member_pmc* for node in *pmc*.

NOTICE

Only PM/Composite context supports this operation.

RETURN VALUES

`PM_SUCCESS`	Success
`EINVAL`	Invalid node number
`EBUSY`	*member_pmc* is associated with another context node is already associated with some context
`ENOSYS`	Operation not supported (*pmc* is not a PM/Composite context)

---

pmAfterSelect(3)

NAME

pmAfterSelect — call after returning from select(2)

SYNOPSIS

int pmAfterSelect(pmContext *pmc);

ARGUMENTS

pmContext *	pmc	IN	pmContext object

DESCRIPTION

`pmAfterSelect()` must be called after returning from `select()` regardless of the return value of `select()`.

RETURN VALUES

`PM_SUCCESS`	Success

---

pmAttachContext(3)

NAME

pmAttachContext — create a context

SYNOPSIS

int pmAttachContext(char *type, int fd, pmContext **pmcp);

ARGUMENTS

char *	type	IN	Type of context
int	fd	IN	File descriptor of context
pmContext **	pmcp	OUT	pmContext object

DESCRIPTION

`pmAttachContext()` creates a pmContext object specified by *type* and *fd* , and attaches it to the calling process.

RETURN VALUES

`PM_SUCCESS`	Success
`ENODEV`	No such device
`EINVAL`	Invalid file descriptor

---

pmBeforeSelect(3)

NAME

pmBeforeSelect — call before select(2)

SYNOPSIS

int pmBeforeSelect(pmContext *pmc);

ARGUMENTS

pmContext *	pmc	IN	pmContext object

DESCRIPTION

`pmBeforeSelect()` must be called before calling `select()`.

NOTICE

After calling `pmBeforeSelect()`, all messages in the receive queue must be extracted by `pmReceive()`.

RETURN VALUES

`PM_SUCCESS`	Success

---

pmErrorString(3)

NAME

pmErrorString — convert an error number to an error string

SYNOPSIS

char *pmErrorString(int error);

ARGUMENTS

int	error	IN	Error number

DESCRIPTION

`pmErrorString()` converts an error number to an error string.

RETURN VALUES

The error string

---

pmExtractNode(3)

NAME

pmExtractNode - extract a member context and member node number

SYNOPSIS

int pmExtractNode(pmContext *pmc, int node, pmContext **member_pmcp, int *member_nodep);

ARGUMENTS

pmContext *	pmc	IN	pmContext object: must be a PM_COMPOSITE type
int	node	IN	Node number in pmc
pmContext **	member_pmcp	OUT	Member context
int *	member_nodep	OUT	Node in member context

DESCRIPTION

`pmExtractNode()` extracts a member context and a member node number of *node* in *pmc*.

NOTICE

Only PM/Composite contexts supports this operation.

RETURN VALUES

PM_SUCCESS	Success
EINVAL	Invalid node number
	No context is associated with node
ENOSYS	Operation not supported (*pmc* is not a PM/Composite context)

---

pmGetContextConfig(3)

NAME

pmGetContextConfig - get configuration of a context

SYNOPSIS

int pmGetContextConfig(pmContext *pmc, pmContextConfig *configp);

ARGUMENTS

pmContext *	pmc	IN	pmContext object
pmContextConfig *	configp	OUT	Context configuration

DESCRIPTION

`pmGetContextConfig()` returns the configuration of a context.

NOTICE

The caller must not free or alter the returned type name string.

RETURN VALUES

PM_SUCCESS	Success

---

pmGetFd(3)

NAME

pmGetFd - extract file descriptors associated with a context

SYNOPSIS

int pmGetFd(pmContext *pmc, int *fd, int *nfdp);

ARGUMENTS

pmContext *	pmc	IN	pmContext object: must be a PM_COMPOSITE type
int *	fd	OUT	File descriptors of context
int *	nfdp	IN	Number of entries in fd array
		OUT	Number of file descriptors

DESCRIPTION

`pmGetFd()` extracts file descriptors associated with context *pmc* for `select()` or `poll()`. It always sets the number of file descriptors to *nfdp and returns `ENOSPC` if the number of array entries is too small.

RETURN VALUES

`PM_SUCCESS`	Success
`ENOSPC`	Number of array entries is too small
`EINVAL`	Number of array entries is less than zero
`ENOMEM`	Not enough memory

---

pmGetMessageQueueStatus(3)

NAME

pmGetMessageQueueStatus - get the message queue status

SYNOPSIS

int pmGetMessageQueueStatus(pmContext *pmc, pmMessageQueueStatus *statp);

ARGUMENTS

pmContext *	pmc	IN	pmContext object
pmMessageQueueStatus *	statp	OUT	Message Queue status

DESCRIPTION

`pmGetMessageQueueStatus()` returns a `pmMessageQueueStatus` structure which contains the number of received messages, sending messages and outstanding remote read operations present or not in the queue.

RETURN VALUES

`PM_SUCCESS`	Success

---

pmGetMtu(3)

NAME

pmGetMtu - get an MTU value of a context

SYNOPSIS

int pmGetMtu(pmContext *pmc, int node, size_t *mtup);

ARGUMENTS

pmContext *	pmc	IN	pmContext object: must be a PM/Composite type
int	node	IN	Node number
size_t *	mtup	OUT	MTU

DESCRIPTION

`pmGetMtu()` returns an `MTU` value of context *pmc* to communicate with node.

RETURN VALUES

`PM_SUCCESS`	Success
`EINVAL`	Invalid node number

---

pmGetMulticastBuffer(3)

NAME

pmGetMulticastBuffer - allocate a buffer for multicasting to nodes

SYNOPSIS

int pmGetMulticastBuffer(pmContext *pmc, int *dest, int ndest, caddr_t *bufp, size_t len);

ARGUMENTS

pmContext *	pmc	IN	pmContext object
int *	dest	IN	Destination node numbers
int	ndest	IN	Number of destination numbers
caddr_t *	bufp	OUT	Buffer address
size_t	len	IN	Buffer length

DESCRIPTION

`pmGetMulticastBuffer()` allocates a message buffer of *len* bytes for multicasting to nodes described in *dest*. If *pmc* is of type PM/Composite and a buffer is allocated, the context is locked until `pmSend()` is called.

NOTICE

Returned buffer addresses can be cast to any type.
Minimum value of len is 1.

RETURN VALUES

`PM_SUCCESS`	Success
`ENOBUFS`	No buffer is available
`EINVAL`	Invalid destination node number
	Invalid length (len < minimum length or len > MTU)
`EBUSY`	Context is already locked
`ENOSYS`	Operation not supported

---

pmGetSelf(3)

NAME

pmGetSelf - get the node number of itself in a context

SYNOPSIS

int pmGetSelf(pmContext *pmc, int *selfp);

ARGUMENTS

pmContext *	pmc	IN	pmContext object
int *	selfp	OUT	Node number of itself

DESCRIPTION

`pmGetSelf()` returns a node number of itself in context *pmc*.

RETURN VALUES

`PM_SUCCESS`	Success
`EINVAL`	Node number is not set

---

pmGetSendBuffer(3)

NAME

pmGetSendBuffer - allocate a buffer to send a message

SYNOPSIS

int pmGetSendBuffer(pmContext *pmc, int dest, caddr_t *bufp, size_t len);

ARGUMENTS

pmContext *	pmc	IN	pmContext object
int	dest	IN	Destination node number
caddr_t *	bufp	OUT	Buffer address
size_t	len	IN	Buffer length

DESCRIPTION

`pmGetSendBuffer()` allocates a buffer to send a *len* bytes of a message, if available. If *pmc* is of type PM/Composite and a buffer is allocated, the context is locked until `pmSend()` is called.

NOTICE

Returned buffer addresses can be cast to any type.
Minimum value of len is 1.

RETURN VALUES

`PM_SUCCESS`	Success
`ENOBUFS`	No buffer is available
`EINVAL`	Invalid destination node number
	Invalid length (len < minimum length or len > MTU)
`EBUSY`	Context is already locked

---

pmIsReadDone(3)

NAME

pmIsReadDone - determine if all previous pmRead()'s are done

SYNOPSIS

int pmIsReadDone(pmContext *pmc);

ARGUMENTS

pmContext *	pmc	IN	pmContext object

DESCRIPTION

`pmIsReadDone()` determines if all previous `pmRead()`'s are done.

RETURN VALUES

`PM_SUCCESS`	All remote reads are done
`EBUSY`	Some remote reads are not yet done
`EIO`	Error occured in some remote reads
`ENOSYS`	Operation not supported

---

pmIsSendDone(3)

NAME

pmIsSendDone - determine if all previous sent messages have been received

SYNOPSIS

int pmIsSendDone(pmContext *pmc);

ARGUMENTS

pmContext *	pmc	IN	pmContext object

DESCRIPTION

`pmIsSendDone()` determines if all previous sent messages have been received by destination nodes.

NOTICE

`pmIsSendDone()` does not guarantee that the messages have been extracted by receiver processes.

RETURN VALUES

`PM_SUCCESS`	All messages have been sent
`EBUSY`	Some messages have not yet been sent
`EIO`	Error occured while sending messages

---

pmIsWriteDone(3)

NAME

pmIsWriteDone - determine if all previous pmWrite()'s are done

SYNOPSIS

int pmIsWriteDone(pmContext *pmc);

ARGUMENTS

pmContext *	pmc	IN	pmContext object

DESCRIPTION

`pmIsWriteDone()` determines if all previous `pmWrite()`'s are done, and all regions to be transferred can be altered.

NOTICE

`pmIsWriteDone()` does not guarantee that the sent data has been written into the remote nodes.

RETURN VALUES

`PM_SUCCESS`	All remote writes are done
`EBUSY`	Some remote writes are not yet done
`EIO`	Error occured in some remote writes
`ENOSYS`	Operation not supported

---

pmMLock(3)

NAME

pmMLock - pin down a specified region and return a locked address handle

SYNOPSIS

int pmMLock(pmContext *pmc, int rmt_node, caddr_t addr, size_t len, pmAddrHandle *hndlp);

ARGUMENTS

pmContext *	pmc	IN	pmContext object
int	rmt_node	IN	Remote node number
caddr_t	addr	IN	Start address of region to be locked
size_t	len	IN	Length of region
pmAddrHandle *	hndlp	OUT	Locked address handle

DESCRIPTION

`pmMLock()` pins down a specified region and returns a locked address handle.

NOTICE

For a composite context *rmt_node* can be specified as `PM_NODE_ANY` to pin-down a region for all remote nodes.
Alignment of addr and len is system dependent.
Minimum value of len is system dependent.

RETURN VALUES

`PM_SUCCESS`	Success
`EINVAL`	Invalid remote node number
	Invalid length
`ENOSPC`	No enough resources to lock a region
`ENOSYS`	Operation not supported

---

pmMUnlock(3)

NAME

pmMUnlock - release a specified pinned-down region

SYNOPSIS

int pmMUnlock(pmContext *pmc, int rmt_node, caddr_t addr, size_t len);

ARGUMENTS

pmContext *	pmc	IN	pmContext object
int	rmt_node	IN	Remote node number
caddr_t	addr	IN	Start address of region to be locked
size_t	len	IN	Length of region

DESCRIPTION

`pmMUnlock()` releases a specified pinned-down region. The actual release of the region may be delayed.

NOTICE

For a composite context *rmt_node* can be specified as `PM_NODE_ANY` for all remote nodes.
Alignment of addr and len is system dependent.
Minimum value of len is system dependent.

RETURN VALUES

`PM_SUCCESS`	Success
`EINVAL`	Invalid remote node number
	Invalid length
`ENOENT`	Region was not locked by `pmMLock()`
`ENOSYS`	Operation not supported

---

pmRead(3)

NAME

pmRead - copy data from a remote region to a local region

SYNOPSIS

int pmRead(pmContext *pmc, int rmt_node, pmAddrHandle rmt_hndl, pmAddrHandle local_hndl, size_t len);

ARGUMENTS

pmContext *	pmc	IN	pmContext object
int	rmt_node	IN	Remote node number
pmAddrHandle	rmt_hndl	IN	Remote address handle
pmAddrHandle	local_hndl	IN	Local address handle
size_t	len	IN	Bytes to read

DESCRIPTION

`pmRead()` copies data from a remote region to a local region.
Both regions must be locked by `pmMLock()`.
Local and remote address handles are (locked address handle + offset).

NOTICE

Alignment of *rmt_hndl*, *local_hndl* and len is system dependent.
Minimum value of len is system dependent.

RETURN VALUES

PM_SUCCESS	Success
EINVAL	Invalid remote node number
	Invalid length
ENOBUFS	No enough resources to read
ENOSYS	Operation not supported

---

pmReceive(3)

NAME

pmReceive - return an address and length of a message

SYNOPSIS

int pmReceive(pmContext *pmc, caddr_t *bufp, size_t *lenp);

ARGUMENTS

pmContext *	pmc	IN	pmContext object
caddr_t *	bufp	OUT	Address of message
size_t *	lenp	OUT	Length of message

DESCRIPTION

`pmReceive()` polls for message arrival and returns an address and a length of the message if available. If *pmc* is of type PM/COMPOSITE and a message is received, the context is locked until `pmReleaseReceiveBuffer()` is called.

NOTICE

Returned buffer address can be cast to any type.

RETURN VALUES

PM_SUCCESS	Message is received
ENOBUFS	No message is received
EIO	Data link level error (CRC error for Myrinet)
EPIPE	Network reset occurred for Myrinet
EBUSY	Context is already locked

---

pmReleaseReceiveBuffer(3)

NAME

pmReleaseReceiveBuffer - release a message buffer

SYNOPSIS

int pmReleaseReceiveBuffer(pmContext *pmc);

ARGUMENTS

pmContext *	pmc	IN	pmContext object

DESCRIPTION

`pmReleaseReceiveBuffer()` releases a message buffer previously received by `pmReceive()`. If *pmc* is of type PM/Composite then the context is unlocked.

RETURN VALUES

`PM_SUCCESS`	Success
`EIO`	Called before a message is received

---

pmRemoveNode(3)

NAME

pmRemoveNode - remove a node

SYNOPSIS

int pmRemoveNode(pmContext *pmc, int node);

ARGUMENTS

pmContext *	pmc	IN	pmContext object: must be a PM/Composite type
int	node	IN	Node number in pmc

DESCRIPTION

`pmRemoveNode()` removes a member node from node in pmc.

NOTICE

Only PM/Composite contexts supports this operation.

RETURN VALUES

PM_SUCCESS	Success
EINVAL	Invalid node number
	No context is associated with node
EIO	Internal error
ENOSYS	Operation not supported

---

pmSend(3)

NAME

pmSend - send a previously allocated message

SYNOPSIS

int pmSend(pmContext *pmc);

ARGUMENTS

pmContext *	pmc	IN	pmContext object

DESCRIPTION

`pmSend()` sends a message previously allocated by `pmGetSendBuffer()` or `pmGetMulticastBuffer()`. If *pmc* is of type PM/Composite then the context is unlocked.

RETURN VALUES

PM_SUCCESS	Success
EIO	Called with no buffer allocated

---

pmTruncateBuffer(3)

NAME

pmTruncateBuffer - truncate a send buffer length

SYNOPSIS

int pmTruncateBuffer(pmContext *pmc, size_t len);

ARGUMENTS

pmContext *	pmc	IN	pmContext object
size_t	len	IN	New buffer length

DESCRIPTION

`pmTruncateBuffer()` truncates a send buffer length previously allocated by `pmGetSendBuffer()` or `pmGetMulticastBuffer()`.

NOTICE

The buffer length cannot be expanded.
Minimum value of len is 1.

RETURN VALUES

`PM_SUCCESS`	Success
`EINVAL`	Invalid length
`EIO`	Called with no buffer allocated

---

pmWrite(3)

NAME

pmWrite - copy data from a local region

SYNOPSIS

int pmWrite(pmContext *pmc, int rmt_node, pmAddrHandle rmt_hndl, pmAddrHandle local_hndl, size_t len);

ARGUMENTS

pmContext *	pmc	IN	pmContext object
int	rmt_node	IN	Remote node number
pmAddrHandle	rmt_hndl	IN	Remote address handle
pmAddrHandle	local_hndl	IN	Local address handle
size_t	len	IN	Bytes to write

DESCRIPTION

`pmWrite()` copies data from a local region to a remote node.
Both regions must be locked by `pmMLock()`.
Local and remote address handles are (locked address handle + offset).

NOTICE

Alignment of *rmt_hndl*, *local_hndl* and *len* is system dependent.
Minimum value of len is system dependent.

RETURN VALUES

PM_SUCCESS	Success
EINVAL	Invalid remote node number
	Invalid length
ENOBUFS	No enough resources to write
ENOSYS	Operation not supported

---

sc_create_temporary_file(3)

NAME

sc_create_temporary_file — create a temporary file

SYNOPSIS

```
#include <sc.h>
int sc_create_temporary_file( char *filename, int *fd );
```

DESCRIPTION

`sc_create_temporary_file()` creates a temporary file in the SCore-D directory. The life time of the created file is the same as the one of the parallel job in which this system call is invoked. Thus, the created file will be deleted when the parallel job terminates.

RETURN VALUES

`sc_create_temporary_file()` always returns 0 if it succeeds. Otherwise it returns the same error number as the Unix `open()` system call.

---

sc_open_temporary_file(3)

NAME

sc_open_temporary_file — open a temporary file

SYNOPSIS

```
#include <sc.h>
int sc_open_temporary_file( char *filename, int *fd );
```

DESCRIPTION

`sc_open_temporary_file()` opens a temporary file in the SCore-D directory.

RETURN VALUES

`sc_open_temporary_file()` always returns 0 if it succeeds. Otherwise it returns the same error number as the Unix `open()` system call.

---

sc_set_monitor(3)

NAME

sc_set_monitor — setting user load monitor

SYNOPSIS

```
#include <sc.h>
void sc_set_monitor(int which, unsigned char value);
```

DESCRIPTION

The `sc_set_monitor()` function set the value which can be viewed via SCore-D load monitor. Currently two monitors which user program can control. If the value of the *which* variable is zero, then value is set for 0th monitor. Otherwise the value is set for first monitor.

RETURN VALUES

None.

---

sc_unlink_temporary_file(3)

NAME

sc_unlink_temporary_file — unlink a temporary file

SYNOPSIS

```
#include <sc.h>
int sc_unlink_temporary_file( char *filename );
```

DESCRIPTION

`sc_unlink_temporary_file()` unlinks a temporary file in the SCore-D directory.

RETURN VALUES

`sc_unlink_temporary_file()` always returns 0 if it succeeds. Otherwise it returns the same error number as the Unix `unlink()` system call.

---

score_become_busy(3)

NAME

score_become_busy — tells SCore-D that the process becomes busy

SYNOPSIS

```
#include <score.h>
void score_become_busy( void );
```

DESCRIPTION

`score_become_busy()` lets SCore-D know the running state of a user process. If a user process becomes busy, this means that the program has just exited from a busy-wait loop waiting for incoming PM messages, and `score_become_idle()` must be called.

---

score_become_idle(3)

NAME

score_become_idle — tells SCore-D that the process becomes idle

SYNOPSIS

```
#include <score.h>
void score_become_idle( void );
```

DESCRIPTION

`score_become_idle()` lets SCore-D know the running state of a user process. If a user process becomes idle, this means that the program is actually waiting for incoming PM messages in a busy-wait loop, then it is preferable to call `score_become_idle()`. By reporting the status of a user process correctly, SCore-D can detect the global state of a user parallel process, and SCore-D can try to de-schedule it if the process is in global idle, or try to kill it if the process is assumed to be globally terminated, possibly because of deadlock.

---

score_get_opt(3)

NAME

score_get_opt — get an SCore option

SYNOPSIS

```
#include <score_options.h>
char *score_get_opt(char *keyword)
```

DESCRIPTION

`score_get_opt()` attempts to get the SCore option from the `scrun` command-line arguments specified by the user.

RETURN VALUES

`score_get_opt()` returns a character string associated with the vale of the keyword argument. `score_get_opt()` returns `NULL` if the keyword is not found. It returns a pointer to a null character if the option is specified but has no associated value.

NOTICE

The SCore options table is initialized in `score_initialize()`. Calling `score_get_opt()` before calling `score_initialize()` results in unpredictable results.

---

score_initialize(3)

NAME

score_initialize — initialize a user process

SYNOPSIS

```
#include <score.h>
void score_initialize( int argc, char **argv );

int score_num_pmnet;
pmContext **score_pmnet;
int score_num_host;
int score_self_host;
int score_num_proc;
int score_self_proc;
int score_num_node;
int score_self_node;
```

DESCRIPTION

`score_initialize()` initializes the parallel process execution environment for a user process. *score_pmnet* is initialized to contain a set of pmContexts. The number of pmContexts is set in the *score_num_pmnet* variable. The user parallel program can communicate with other nodes using the pmContext(s) stored in *score_pmnet*. The device type of the pmContext is always PM/Composite, so that PM communication operation is mutually exclusive on a SMP host.

The number of hosts and host number (ID) in the parallel process are set in *score_num_host* and *score_self_host*, respectively. The number of processes in the host where the user program is running and its identification are set in *score_num_proc* and *score_self_proc*, respectively. The number of nodes and node identification of the parallel process are set in *score_num_node* and *score_self_node*, respectively.

NOTICE

Output to standard output is NOT allowed prior to calling `score_initialize()`.

---

gather(6)

NAME

gather — Gather data from the cluster

SYNOPSIS

gather [-node *node_number*] [-file *filename*]

DESCRIPTION

`gather` must be invoked by the `scrun`) command. It gathers standard input of each gather process running on the cluster nodes and forwards them to the standard output of `scrun`.

The options available for gather:

-node *node_number* Only the standard input specified for node *node_number* is forwarded, otherwise all the standard inputs are gathered sequentially starting from node 0 and output to scrun

-file *filename* The corresponding file on the cluster hosts is gathered. If *filename* does not start with "/", then it is assumed that the file is located in a temporary directory under the `/var/scored` directory. If this option is not specified, then the standard inputs of each process are gathered

---

NAME

scatter - Broadcast standard input of scrun

SYNOPSIS

scatter [-node *node_number*] [-file *filename*]

DESCRIPTION

`scatter` must be invoked by the `scrun` command. It broadcasts standard input of the `scrun` process to all `scatter` processes running on a cluster.

The options available for scatter:

-node *node_number* scrun standard input is only copied to the node specified by *node_number* . If this option is not specified then standard input of scrun is broadcast to all scatter processes running on a cluster

-file *filename* The broadcasted data is written to the file *filename* . If *filename* does not start with "/", then a temporary output file is created under the /var/scored directory, and when a parallel job finishes, this file is automatically deleted. If the `-file` option is not specified, then the broadcasted data is output to the standard output of each `scatter` process on the cluster. Note that when `scatter` is executed with the `-file` option on an SMP cluster, then only one output file is created on each host.

The `scatter` command is designed with the view to be used with the SCore-D parallel job management facility (see also `scrun` man page). `scatter` copies a file out of a cluster into cluster hosts in a scalable way. Or, users can run normal sequential programs or Unix commands run on a cluster in parallel using the `system` SCore-D command.

Here are some examples:

```
$ scrun -nodes=8 scatter == a.out < my.dat
```

In this example, the standard input file my.dat is fed as a standard input for all processes of `a.out` running on a cluster.

```
$ scrun -nodes=16x1 scatter -file cluster.dat :: a.out < local.dat
```

In this case, the local file local.dat is copied to the file `cluster.dat` in a temporary directory under `/var/scored` on all 16 hosts, and then the user program a.out is executed on a cluster so that the `a.out` program can read the broadcasted file.

SEE ALSO

scrun(1), system(6), gather(6)

system(6)

NAME

system - Execute a sequential program on the cluster in parallel

SYNOPSIS

system [-host] *seqprog* [*arg* ...]

DESCRIPTION

system must be invoked by the scrun command. It executes the sequential program, *seqprog* , on the cluster by calling execvp(3) on each node. Arguments to *seqprog* should be included after the program.

The options available for system:

-host Only one process on a host executes the sequential program. Other processes terminate immediately

The system command is designed to execute the sequential command in parallel on a cluster. To distinguish from other processes, environment variables, `SCORE_SELF_PROC`, `SCORE_SELF_HOST`, `SCORE_SELF_NODE`, `SCORE_NUM_PROC`, `SCORE_NUM_HOST`, and `SCORE_NUM_NODE` are set the appropriate value.

Upon execution, the current working directory is set to a temporary directory created each time a parallel job is run. The temporary directory is located somewhere in the `/var/scored` directory on each compute host. By doing this, the temporary file created by the `scatter` program can be read by the sequential program on each host. There are library functions, `sc_create_temporary_file()` and `sc_open_temporary_file()` to create or access the temporary file without regarding to the exact pathname.

---

score_compiler_list(8)

NAME

score_compiler_list - display SCore backend compiler list

SYNOPSIS

score_compiler_list -script *script* [-check] [-path] [-noalias] [-default] score_compiler_list -env *environ* [-check] [-path] [-noalias] score_compiler_list -all [-check] [-path] [-noalias]

DESCRIPTION

score_compiler_list prints backend compiler information. If you specify -script *script* option, *script* 's backend compilers are printed. If you specify -env *environ* , specific scripts for *environ* environment backend compiler information are printed. If you specifies -all, all scripts backend compiler are printed.

For example, score_compiler_list -script mpicc output as follows:

```
gnu
pgi
```

This meens you can specify gnu and pgi compiler on mpicc. If you specify -noalias option, score_compiler_list don't print aliases. If you specify -default option, score_compiler_list print only default compiler. If you specify -path option, score_compiler_list print compiler path as follows:

```
gnu:gcc
pgi:pgcc
```

If you specify -check option, score_compiler_list test the backend compilers commands are existed, and print result.

EXAMPLES

```
% score_compiler_list -script mpicc
% score_compiler_list -all -check
```

Index