Scali Library User’s Guide

Acknowledgement

The development of ScaMPI has benefited greatly from the work of people not connected to Scali. We wish especially to thank the developers of MPICH for their work which served as a reference when implementing the first version of ScaMPI.
The list of persons contributing to algorithmic ScaMPI improvements is impossible to compile here. We apologise to those who remain unnamed and mention only those who certainly are responsible for a step forward.

Scali is thankful to Rolf Rabenseifner for the improved reduce algorithm used in ScaMPI.

Table of contents

Chapter 1 Introduction

A Scali System is a set of SCI interconnected nodes, where each node is a multi-processor workstation running Linux. To help you use the full potential power of such a system we have developed a suite of libraries.

1.1 Scali Library Suite

1.1.1 ScaMPI

ScaMPI is a high performance MPI implementation. The programming environment for ScaMPI provides a variety of options and tools for tuning and debugging. ScaMPI utilises shared memory on intranode communication, and the fast SCI interconnect on internode communication. Any parallel MPI-conforming application can be run with ScaMPI and benefit from the SCI performance.

The chapters conserning ScaMPI is written for users which have a basic understanding of MPI [1, 2, 3], and some basic knowledge of the C and/or Fortran programming language.
gcc and **bash** are used for all examples.

1.1.2 ScaShmem

ScaShMem is an implementation of the Cray/SGI Shmem abstraction. It is built on top of ScaMPI

1.1.3 ScaIP

ScaIP is a version of IP based on SCI.

1.1.4 ScaMAC

ScaMAC, Scali Media Access Control driver for SCI, includes a kernel mode driver and some utilities to allow fast transfer of data on SCI.

Chapter 2 Using ScaMPI

This chapter describes the setup, compile, link and run of a program using ScaMPI. Furthermore some useful tools for debugging and profiling are briefly discussed.

Please note that the “ScaMPI release notes” are available in the **/opt/scali/doc/ScaMPI** directory.

2.1 Setting up a ScaMPI environment

2.1.1 ScaMPI environment variables

The use of ScaMPI requires that some environment variables are defined. These are usually set in the standard startup scripts (e.g. .**bashrc** when using **bash**), but they can also be defined manually.

Name	Description
MPI_HOME	Installation directory. For a standard installation, the variable should be set as: **`export MPI_HOME=/opt/scali`**
LD_LIBRARY_PATH	Path to dynamic link libraries. Must be set to include the path to the directory where these libraries can be found: **`export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:$MPI_HOME/lib`**
PATH	Path variable. Must be updated to include the path to the directory where the MPI binaries can be found: **`export PATH=${PATH}:$MPI_HOME/bin`**

Table 2-1: Environment variables

Normally, the ScaMPI library’s header files **mpi.h** and **mpif.h** reside in the $MPI_HOME/include directory.

2.2 Compiling and linking

MPI is an “Application Programming Interface”(API) and not an “Application Binary Interface”(ABI). This means that you as a main rule should recompile and relink your application when starting to use ScaMPI. But since the MPICH-implementation is

widely used we have made ScaMPI ABI compatible depending on versions of MPICH and ScaMPI. Please check “ScaMPI release notes” for details. Having an application linked with mpich you should be able to just change library-path if it is dynamically linked or you have to relink it if you have linked it statically.

2.2.1 Compiler support

ScaMPI is a C++ library built using the GNU compiler.This implies that you have to link with the GNU runtime library. Depending on the compiler used, the way to link with the ScaMPI libraries varies. Check the ”ScaMPI release notes” for information on supported compilers and how linking is done. Please note that the GNU compiler, or a similar version of the C++ compiler, must be installed on your system. The GNU compilers are included in the ScaFegcs package, available for download at **http://www.scali.com/download**.

2.2.2 Compiler flags

The following string *must* be included as compile flags (**bash** syntax):

```
“-D_REENTRANT -I$MPI_HOME/include”
```

2.2.3 Linker flags

The following string outlines the setup for the necessary link flags (**bash** syntax):

```
“-L/opt/scali/lib $CRT_BEGIN -lmpi $CRT_END”
```

The runtime setup CRT_BEGIN and CRT_END libraries are defined for some compilers. Please note that when linking a Fortran main program, the Fortran interface library **libfmpi** must be included *before* CRT_BEGIN.

2.3 Running ScaMPI programs

Note that executables issuing ScaMPI calls ***cannot*** be started directly from a shell prompt. ScaMPI programs can either be started using the MPI monitor program **mpimon**, the wrapper script **mpirun**, or from the Scali Universe GUI [4].

2.3.1 Naming convention

When an application program is started, ScaMPI is modifying argv[0]. The following convention is used for the executable, reported on the command line using the Unix utility **ps**:

<userprogram>-<rank number>(mpi:<pid>@<nodename>)

where:

<userprogram> is the name of the application program.
<rank number> is the application's mpi-process rank number.
<pid> is the Unix process identifier of the monitor program **mpimon**.
<nodename> is the name of the node where **mpimon** is running.

Note that ScaMPI requires a homogenous file system image, i.e., a file system providing the same path and program names on all nodes of the Scali System.

2.3.2 mpimon - monitor program

The control and start-up of an ScaMPI application is monitored by **mpimon**. The program **mpimon** has several options which can be used for optimising ScaMPI performance. Normally it should not be necessary to use any of these options. However, unsafe MPI programs [3] might need buffer adjustments to solve deadlocks. Trading performance by changing communication space is best avoided if there is no compelling reason to do so.

2.3.2.1 Basic usage

Normally the program is invoked as:

mpimon <userprogram> <programoptions> -- <nodename> [<count>] [<nodename> [<count>]]...

Parameter	Description
<userprogram>	Name of application program.
<programoptions>	Program options for the application program.
--	Separator, marks end of user program options.
[<nodename> [<count>]]	Name of node and the number of mpi-processes to run on that node. The option can occur several times in the list. Mpi-processes will be given ranks sequentially according to the list of node-number pairs.

Table 2-2: Basic options to **mpimon**

2.3.2.2 Advanced usage

The complete syntax for the program:

mpimon [<mpimon-option>]... <program & node-spec> [-- <program & node-spec>]...

Parameter	Description
<program & node-spec>	**<program spec> -- <node spec> [<node spec>]...**
<program spec>	**<userprogram>[<programoptions>]...**
<userprogram>	Name of application program.
<programoptions>	Program options for the application program.
--	Separator, signals end of user program options.
<node spec>	**<nodename> [<count>]**
<nodename>	Name of node. Given either as a node name or as an Internet address expressed in the Internet standard dot notation.
<count>	Number of mpi-processes to run on node. If **<count>** is omitted, one mpi-process is started on each node specified.

Table 2-3: mpimon parameters

Numeric values can be given as **mpimon** options in the following way:

Option	Description
<numeric value>	**<decimal value> | <decimal value><postfix>**
<postfix>	**<K>: <numeric value> = <decimal value> * 1024** **<M>: <numeric value> = <decimal value> * 1024 * 1024**

Table 2-4: Numeric input

A complete lists of available **mpimon** options:

mpimon option	Description
--	Separator, marks end of user program options.
-automatic <*selection*>	Set automatic-mode for process(es). Default: **none** Legal: **'n,m,o..' = (list) or 'n-m' = (range) or 'all'**
-barrier_fanin <*count*>	Set number of barrier fanin reads. Default: **8**
-barrier_fanout <*count*>	Set number of barrier fanout reads. Default: **8**
-debug <*selection*>	Set debug-mode for process(es). Default: **none** Legal: **'n,m,o..' = (list) or 'n-m' = (range) or 'all'**
-debugger <*debugger*>	Set debugger to start in debug-mode.
-disable-timeout	Disable process timeout.
-display <*display*>	Set display to use in debug-/manual-mode.
-dryrun <*mode*>	Set dryrun-mode. Default: **none** Legal: **'none' or 'totalview'**
-environment <*value*>	Define how to export environment. Default: **export** Legal: **'export' = all or 'mpi' = MPI_?? or 'none'**
-exact_match	Set exact-match-mode.
-execpath <*execpath*>	Set path to internal executables.
-help	Display available options.
-home <*directory*>	Set installation-directory.
-immediate_handling <*selection*>	Handling of immediates. Default: **lazy** Legal: **lazy, threaded, automatic**
-inherit_limits	Inherit userdefinable limits to processes.
-init_comm_world	Initialise MPI_COMM_WORLD at startup (all channels are created).

Table 2-5: Complete list of **mpimon** options

mpimon option	Description
-inter_adapters <*adapters*>	Set list of sci adapters for inter-communication. Default: **all** Legal: **'n,m,o..' = (list) or 'n-m' = (range) or 'all'**
-inter_channel_inline_threshold <*size*>	Set threshold for inlining (in bytes) per inter-channel. Default: **1030**
-inter_channel_size <*size*>	Set buffer size (in bytes) per inter-channel. Default: **4K** Legal: **Powers of 2**
-inter_chunk_size <*size*>	Set chunk-size for inter-communication. Default: **512k** Legal: **Multiplum of pages**
-inter_eager_count <*count*>	Set number of buffers for eager inter-protocol. Default: **2**
-inter_eager_size <*size*>	Set buffer size (in bytes) for eager inter-protocol. Default: **256K** Legal: **Powers of 2**
-inter_eager_threshold <*size*>	Set threshold (in bytes) for eager inter-protocol. Default: **32K** Legal: **Powers of 2**
-inter_pool_size <*size*>	Set buffer-pool-size for inter-communication. Default: **32M** Legal: **Multiplum of pages**
-inter_transporter_count <*count*>	Set number of buffers for transporter inter-protocol. Default: **4** Legal: **Powers of 2**
-inter_transporter_size <*size*>	Set buffer size (in bytes) for transporter inter-protocol. Default: **256K**
-intra_channel_inline_threshold <*size*>	Set threshold for inlining (in bytes) per intra-channel. Default: **560**
-intra_channel_size <*size*>	Set buffer size (in bytes) per intra-channel. Default: **4K** Legal: **Powers of 2**
-intra_chunk_size <*size*>	Set chunk-size for intra-communication. Default: **1M** Legal: **Multiplum of pages**
-intra_eager_count <*count*>	Set number of buffers for eager intra-protocol. Default: **2**

Table 2-5: Complete list of **mpimon** options

mpimon option	Description
-intra_eager_size <*size*>	Set buffer size (in bytes) for eager intra-protocol. Default: **128K** Legal: **Powers of 2**
-intra_eager_threshold <*size*>	Set threshold (in bytes) for eager intra-protocol. Default: **1M** Legal: **Powers of 2**
-intra_pool_size <*size*>	Set buffer-pool-size for intra-communication. Default: **4M** Legal: **Multiplum of pages**
-intra_transporter_count <*count*>	Set number of buffers for transporter intra-protocol. Default: **4** Legal: **Powers of 2**
-intra_transporter_size <*size*>	Set buffer size (in bytes) for transporter intra-protocol. Default: **64K**
-manual <*selection*>	Set manual-mode for process(es). Default: **none** Legal: **'n,m,o..' = (list) or 'n-m' = (range) or 'all'**
-read <*file*>	Read parameters from the named file. Default: **none**
-separate_output <*selection*>	Enable separate output for process(es). Filename: ScaMPIoutput_host_pid_rank Default: **none** Legal: **'n,m,o..' = (list) or 'n-m' = (range) or 'all'**
-shmem	Application use Cray ShMem library.
-sm_debug <*selection*>	Set debug-mode for submonitor(s). Default: **none** Legal: **'n,m,o..' = (list) or 'n-m' = (range) or 'all'**
-sm_manual <*selection*>	Set manual-mode for submonitor(s). Default: **none** Legal: **'n,m,o..' = (list) or 'n-m' = (range) or 'all'**
-sm_trace <*selection*>	Enable trace for submonitor(s). Default: **none** Legal: **'n,m,o..' = (list) or 'n-m' = (range) or 'all'**
-statistics	Enable statistics.
-stdin <*selection*>	Distribute standard in to process(es). Default: **none** Legal: **'n,m,o..' = (list) or 'n-m' = (range) or 'all'**

Table 2-5: Complete list of **mpimon** options

mpimon option	Description
-timeout <*timeout*>	Set timeout (elapsed time in seconds) for run. Legal: **Positive number**
-trace <*selection*>	Enable trace for process(es). Default: **none** Legal: **'n,m,o..' = (list) or 'n-m' = (range) or 'all'**
-verbose	Display values for user-options.
-Version	Display version of monitor.
-working_directory <*directory*>	Set working directory.
-xterm <*xterm*>	Set xterm to use in debug-/manual-mode.

Table 2-5: Complete list of **mpimon** options

2.3.3 mpirun - wrapper script

mpirun is a wrapper script for **mpimon**, giving MPICH [10] style startup for ScaMPI applications. Instead of the **mpimon** syntax, where a list of pairs of node name and number of mpi-processes is used as startup specification, **mpirun** uses only the total number of mpi-processes.

Using **scaconftool** (see [4]), **mpirun** attempts to generate a list of operational nodes. Note that only operational nodes are selected. If no operational node is available, an error message is printed and **mpirun** terminates. If **scaconftool** is not available, **mpirun** attempts to use the file /**opt/scali/etc/ScaConf.nodeidmap** for selecting the list of operational notes. In the generated list of nodes, **mpirun** evenly divides the mpi-processes among the nodes.

2.3.3.1 mpirun usage

mpirun <mpirunoptions> <mpimonoptions> <userprogram> [<programoptions>]

Parameter	Description
<mpirunoptions>	**mpirun** options
<mpimonoptions>	Options passed on to **mpimon**
<userprogram>	Name of application program to run.
<programoptions>	Program options passed on to the application program.

Table 2-6: mpirun format

mpirun option	Description
-np *<count>*	Total number of mpi-processes to be started, default 2.
-npn *<count>*	Maximum number of mpi-processes pr. node, default **np** <count>/nodes.
-pbs	Submit job to PBS queue system
-pbsparams *<"params">*	Specify PBS scasub parameters
-p4pg *<pgfile>*	Use mpich compatible pgfile for program, mpi-process and node specification. pgfile entry: <nodename> <#procs> <progname> Program name given at command line is additionally started with one mpi-process at first node
-v	Verbose.
-gdb	Debug all mpi-processes using the GNU debugger **gdb**.
-maxtime|-cpu *<time>*	Limit runtime to <time> minutes.
-machinefile *<filename>*	Take the list of possible nodes from <filename>
-noconftool	Do not use **scaconftool** for generating nodelist.
-noarchfile	Ignore the **/opt/scali/etc/ScaConf.nodearchmap** file (which describes each node).
-H *<frontend>*	Specify nodename of front-end running the **scaconf** server.
-mstdin *<proc>*	Distribute **stdin** to mpi-process(es). *<proc>*: all (default), none, or mpi-process number(s).
-part *<part>*	Use nodes from partition <part>

mpirun option	Description
-q	Keep quiet, no **mpimon** printout.
-t	test mode, no mpi program is started
<*params*>	Parameters not recognized are passed on to **mpimon**.

Table 2-7: mpirun options

2.4 Useful tools

Debugging with a separate debugging session for each mpi-process requires no parallel debugger. However, debugging several mpi-processes in separate debugging sessions, may become a time consuming and tedious task.

2.4.1 Debugging with a sequential debugger

ScaMPI applications can be debugged using a sequential debugger. By default, the GNU debugger **gdb** is invoked by **mpimon**. If another debugger is to be used, specify the debugger using the **mpimon** option **-debugger <debugger>.**

To set debug-mode for one or more mpi-processes, specify the mpi-process(es) to debug using the **mpimon** option **-debug <select>**. In addition, note that the **mpimon** option **-display <display>** should be used to set the display for the **xterm** terminal emulator. An **xterm** terminal emulator, and one debugger, is started for each of the mpi-processes being debugged.

For example, to debug two mpi-processes with rank 0 and 1 using the default **gdb** debugger:

mpimon -display my_pc:0.0 -debug 0,1 <program & node spec>

Initially, for both mpi-process 0 and mpi-process 1, an **xterm** window is opened. Next, in the upper left hand corner of each **xterm** window, a message containing the application program's run parameter(s) is displayed. Typically, the first line reads **Run parameters: *run <programoptions>***. The information following the colon, i.e., ***run <programoptions>*** is needed by both the debugger and the ScaMPI application being debugged. Finally, one debugger is started for each session. In each debugger's **xterm** window, do whatever debugging action that is appropriate before the mpi-process is started. Then, when ready to run the mpi-process, paste the ***run <programoptions>*** into the debugger to start running.

2.4.2 Useful built-in-tools for debugging

2.4.2.1 Using built-in segment protect violation handler

If you have an application that terminates with a SIGSEGV-signal it is often useful to be able to freeze the situation instead of exiting which is normal behaviour. The built-in SIGSEGV-handler can be made to do this by defining the environment-variable SCAMPI_INSTALL_SIGSEGV_HANDLER. Legal option are:

1 The handler dump all registers and start looping. Attaching with a debugger will then give the possibility to examine the situation giving the segment protect violation.

2 the handler dump of registers but all processes will exit afterwards.
All other values will disable the installation of the handler.

2.4.2.2 Using built-in sanity-check of data.

If you are unsure of the quality of you SCI-network is nice to be able to run ScaMPI with extra sanity-checking of data. This is a slower mode where we make a checksum of all data both at the sender and the receiver and compare. This mode is controlled by the environment-variable SCAMPI_DATACHECK_ENABLE and has the following options:
1 when error is detected report and loop
2 when error is detected report and exit
All other value disables the sanity check.

2.4.3 Profiling ScaMPI applications

When developing MPI programs it is difficult to do performance analysis.
There are different tools available that can be useful in detecting / analysing performance bottlenecks:

- ScaMPI has built-in proprietary trace and profiling tools
- Freeware that uses the standard MPI profiling interface such as MPE which is developed by the MPICH-implementors. It is part of a MPICH-distribution and we have also made it available as a part of ScaMPI-distribution
- Commercial tools that collect information during a run and postprocesses and presents afterwards. One example of this is Vampir from Pallas GmbH. See **http://www.pallas.de** for more information.

The main difference between these tools is that the ScaMPI tools can be used with an existing binary while the other tools require reloading with extra libraries.

2.4.3.1 Using ScaMPI built-in trace

To use built-in trace-facility you need to set the environment-variable SCAMPI_TRACE specifying what options you want to apply. The following options can be specified: (<...-list> is a semicolon-separated list of Posix-regular-expressions.)

Name	Description
-b	Trace beginning and end of each MPI_call
-s <seconds>	Start trace after <seconds> seconds
-S <seconds>	End trace after <seconds> seconds

Table 2-8: Options for SCAMPI_TRACE

Name	Description
-c <calls>	Start trace after <calls>MPI_calls
-C <calls>	End trace after <calls>MPI_calls
-p <selection>	Enable for process(es): 'n,m,o..' = (list) or 'n-m' = (range) or 'all'
-t <call-list>	Enable for MPI_calls in <call-list>. MPI_call = 'MPI_call' | 'call'
-x <call-list>	Disable for MPI_calls in <call-list>. MPI_call = 'MPI_call' | 'call'
-f <format-list>	Define format: 'timing', 'arguments', 'rate'
-v	Verbose
-h	Print this list of options

Table 2-8: Options for SCAMPI_TRACE

By default only one line is written per MPI-call. The "-b" option is useful when trying to pinpoint what MPI-call that are started but not completed (deadlocks). The "-s/-S/-c/-C" -options are nice to have if you have an application that runs ok for a longer period and then stop or if you want to have a closer look at some part of the execution of the application.
From time to time it is feasible to trace only one or a few of the processes. Specifying the "-p" options allows you to pick the processes you want to trace.
All MPI-calls are enabled for tracing by default. If you want to look only on a few calls you could do that by specifying a "-t <call-list>" option or if you want to exclude some call you add a "-x <call-list>" option. The "-t" will disable all tracing and then enable those calls that matches the <call-list>. The matching is done using "regular-posix-expression"-syntax. "-x" will to the opposite; First enable all tracing and then disable those call matching <call-list>.
Examples:
"-t MPI_Irecv" : Trace only immediate recv (MPI_Irecv)
"-t isend;irecv;wait" :Trace only MPI_Isend, MPI_Irecv and MPI_Wait
"-t MPI_[b,r,s]*send" : Trace only send-calls (MPI_Send, MPI_Bsend, MPI_Rsend, MPI_Ssend)
"-t i[a-z]*" : Trace only calls beginning with MPI_I
As you can see calls can be specified with or without the "MPI_"-prefix. You can also use upper- or lower-case when specifying calls. The default format of the output has the following parts:
<absRank>: <MPIcall><commName>_<rank><call-dependant-parameters>
where

Field	Description
<absRank>	is rank within MPI_COMM_WORLD
<MPIcall>	is name of MPI-call
<commName>	is name of communicator
<rank>	is rank within communicator used

Table 2-9: Fields in output from built-in trace

This format can be extended by using the "-f"-option. Adding "-f arguments" will give some more information concerning length of messages. If "-f timing" is given you get some timing-info between the <absRank>and <MPIcall>-fields. It has the following format:
+<relSecs> S <eTime>
where
"-f rate" will add some rate-information. The rate is calculated by dividing the number-

Name	Description
<relSecs>	is elapsed time in seconds since returning to the application from MPI_Init
<eTime>	is elapsed execution time for current call

Table 2-10: Timespec in output from built-in trace

of-bytes transferred by the elapsed time to execute the call. All parameters to -f can be abbreviated and can occur in any mix.
The verbose-option(-v) will print information about which options you have selected.

Normally you will not get any error-messages concerning the options you have given. But if you add -verbose as command-line option to mpimon, errors will be printed.

Rate-measurements on Alpha-processors will be inaccurate due to low-resolution-timers.

2.4.3.2 Using ScaMPI built-in timing

To use the built-in timing you need to set the environment variable SCAMPI_TIMING specifying what options you want to apply.
The following options can be specified:
(<...-list> is a semicolon-separated list of Posix-regular-expressions.)

Name	Description
-s <seconds>	Print for intervals of <seconds>seconds
-c <calls>	Print for intervals of <calls>MPI_calls
-p <selection>	Enable for process(es) 'n,m,o..' = (list) or 'n-m' = (range) or 'all'
-f <call-list>	Print after MPI_calls in <call-list>: MPI_call = 'MPI_call' | 'call'
-v	Verbose
-h	Print this list of options

Table 2-11: Options for SCAMPI_TIMING

Printing of timing-information can be either at a fixed time-interval if you specify "-s <seconds>" or for a fixed number-of-calls-interval if you use "-c <calls>". You can also get output after specific MPI-calls if using "-f <call-list>"; See above for details how to write <call-list>.
The output has two parts; First a timing-part followed by a buffer-statistics-part. The first part has the following layout:
All lines starts with **<rank>:** where <rank>: is rank within MPI_COMM_WORLD. This part is included to facilitate separation of output (grep).
The rest of the format has the following fields:
<MPIcall><Dcalls><Dtime><Dfreq> <Tcalls><Ttime><Tfreq>
where

Name	Description
<MPIcall>	is name of MPI-call
<Dcalls>	is number of calls to <MPIcall> since last printout
<Dtime>	is sum of execution-time for calls to <MPIcall> since last printout

Table 2-12: Fields in output from built-in timing

Name	Description
<Dfreq>	is average time-per-call for calls to <MPIcall> since last print-out
<Tcalls>	is number of calls to <MPIcall>
<Ttime>	is sum of execution-time for calls to <MPIcall>
<Tfreq>	is average time-per-call for calls to <MPIcall>

Table 2-12: Fields in output from built-in timing

After all detail-lines (one per MPI-call which has been called since last printout), there will be a line with the sum for all calls followed by a line giving the overhead introduced when obtaining the timing-measurements.
The second part containing the buffer-statistics has two types of lines; one for receives and one for sends.
"Receive-lines" has the following fields:
<Comm><rank> recv from <from>(<worldFrom>):<commonFields>
where

Name	Description
<Comm>	is communicator being used
<rank>	is rank within <Comm>
<from>	is rank within <Comm>
<worldFrom>	is rank within MPI_COMM_WORLD

Table 2-13: Fields in”recv”-lines from built-in timing

"Send-lines" has the following fields:
<Comm><rank> send to <to>(<worldTo>):<commonFields>
where

Name	Description
<Comm>	is communicator being used
<rank>	is rank within <Comm>
<to>	is rank within <Comm>
<worldTo>	is rank within MPI_COMM_WORLD

Table 2-14: Fields in"send"-lines from built-in timing

The **<commonFields>** are as follows:
!<count>!<avrLen>!<zroLen>!<inline>!<eager>!<transporter>!
where

Name	Description
<count>	is number of sends/receives
<avrLen>	is average length of messages in bytes
<zroLen>	is number of messages sent/received using zero-bytes-mechanism
<inline>	is number of messages sent/received using inline-mechanism
<eager>	is number of messages sent/received using eagerbuffer-mechanism
<transporter>	is number of messages sent/received using transporter-mechanism

Table 2-15: Common filelds in output from built-in timing

For more details on the different mechanisms, see "ScaMPI Description".

Timing-measurements on Alpha-processors will be inaccurate due to low-resolution-timers.

2.4.3.3 Using ScaMPI built-in cpu-usage

To use built-in cpu-usage-timing you need to set the environment variable SCAMPI_CPU_USAGE.
The information displayed is collected with the system-call "times"; see man-pages for more information.
The output has two different blocks. The first block contains cpu-usage by the submonitors on the different nodes. One line is printed for each submonitor followed by a sum-line and an average-line. The second block consists of one line per process followed by a sum-line and an average-line.

2.4.4 Profiling with ScaMPE

The ScaMPE libraries are adapted versions of the MPE libraries from MPICH [10]. An executable program linked with one of the ScaMPE libraries **libtmpi, liblmpi** or **libampi** collects performance data during runtime. Normally, the libraries are installed in the directory **/opt/scali/contrib/lib**, and the **upshot** tool, described below, is installed in **/opt/scali/contrib/bin**.

The main components of ScaMPE are:
- A set of routines for creating logfiles for examination by the visualization tool **upshot**.
- Trace or real time animation of MPI calls.
- A shared display parallel X graphics library.

2.4.4.1 Linking an ScaMPI application

Profiling using one of the ScaMPE libraries is achieved by linking with the appropriate ScaMPE library *before* the standard ScaMPI library **libmpi**.

- Trace MPI calls - library **libtmpi**
  To trace all MPI calls, apply **-ltmpi**. Each MPI call is preceded by a line that contains the rank in MPI_COMM_WORLD of the calling process, and followed by another line indicating that the call has completed. Most send and receive routines also indicate the values of count, tag, and partner (destination for sends, source for receives). Output is to standard output **stdout**.

- Generate log file - library **liblmpi**
  To generate an **upshot** style log file of all MPI calls, apply **-llmpi**. When the application is about to finish, an information message is printed to **stdout**, and the trace data is written to a log file for post-processing. The name of the log file, with suffix *.alog*, is created based on the argument provided in argv[0]. Note that when an application program is started, ScaMPI is modifying argv[0], as described in section 2.3. However, the log file name is always created as **executablename-<ScaMPI postfix>.alog**. For example, if the program being

profiled is **sendrecv**, the generated log file is **sendrecv-<ScaMPI postfix>.alog**.

- Real time animation - library **libampi**
  To produce a real-time animation of the program, apply **-lampi -lmpe -lm -lX11**. Note that this requires the MPE graphics in **libmpe**, and that X11 Window System operations are used. To link the X11 libraries (**libX11**), it may be necessary to provide a specific path for the libraries. In addition, note that to resolve some mathematics references used, the standard library **libm** must be included in the link command line. For a description of the MPE graphic routines, see the MPICH documentation [10].

Notes for Fortran users
For a Fortran program, it is necessary to include the Fortran wrapper library **libfmpi** ahead of the profiling libraries. This allows C routines to be used for implementing the profiling libraries for use by both C and Fortran programs. For example, to generate an **upshot** style log file in a Fortran program, the libraries are included in the order **-lfmpi -llmpi -lm**.

2.4.4.2 Examine the generated log file - upshot

To examine a log file generated using **liblmpi**, the parallel program visualization tool **upshot** can be used to analyse the program performance. Note that **upshot** uses the environment variable $DISPLAY to select the display to use.

Start the visualization tool:

```
/opt/scali/contrib/bin/upshot
```

When started, browse and select the appropriate log file to be analysed. For more information, see the document named README_UPSHOT in the directory **/opt/scali/contrib/doc/ScaMPE**.

If **upshot** is not available, any other visualization tool, e.g., **nupshot**, that understands the log file format can be used instead. For more information, see the MPICH documentation [10].

2.5 An example program

When the ScaMPItst package has been installed, the source code and the executable code, for both the **hello-world** example program and a number of test programs, are located under the **/opt/scali/examples/src** and the **/opt/scali/examples/bin** directories. A description of each program in the package can be found in the README file, located in the **/opt/scali/doc/ScaMPItst** directory.

As examples, the MPI program named **hello-world** is used. It exists as a C program in the file **hello-world.c**, and as a Fortran program in the file **hello-world.f.** They are compiled and linked using GNU compilers. Before compilation, it is assumed that the BASH shell environment variable has been properly defined. In addition, ScaMPI must have been installed and function correctly.

2.5.1 Hello-world.c - source in C

```
#include <stdio.h>
#include “mpi.h”

void main(int argc, char** argv)
{
      int rank;
      int size;
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      printf(“Hello-world, I'm rank %d; Size is %d\n”, rank, size);
      MPI_Finalize();
```

2.5.2 Hello-world.f - source in Fortran

```
      program hello_world

implicit none
include 'mpif.h'
integer rank,size,ierr

call mpi_init(ierr);
call mpi_comm_rank(MPI_COMM_WORLD,rank,ierr);
call mpi_comm_size(MPI_COMM_WORLD,size,ierr);
write (*,'(A,I3,A,I3)') "Hello-world, I'm rank ",rank,
& “; Size is “,size
call mpi_finalize(ierr);
end
```

2.5.3 Compiling

```
% gcc -c -D_REENTRANT -I$MPI_HOME/include hello-world.c
% g77 -c -D_REENTRANT -I$MPI_HOME/include hello-world.f
```

2.5.4 Linking

```
% gcc hello-world.o -L$MPI_HOME/lib -lmpi -o hello-world
% g77 hello-world.o -L$MPI_HOME/lib -lfmpi -lmpi -o hello-world
```

2.5.5 Running

Start the **hello-world** program on the 3 nodes named nodeA, nodeB and nodeC.

```
% mpimon hello-world -- nodeA 1 nodeB 1 nodeC 1
```

The **hello-world** program should produce the following output:

```
Hello-world, I'm rank 0; Size is 3
Hello-world, I'm rank 1; Size is 3
Hello-world, I'm rank 2; Size is 3
```

2.6 MPI test programs

The ScaMPItst package contains a collection of MPI test programs for ScaMPI. The following sections give a brief description of some of the test programs, which can be used to measure basic MPI performance. To re-compile any of the test programs, you may use the included *Makefile* found in the appropriate **/opt/scali/examples/src** directory.

2.6.1 Producer - a producer-consumer MPI test program

Producer is a simple producer-consumer program. Mpi-processes with rank 0, 1, 2, ..., n/2-1 send data while mpi-process n/2, n/2+1, ...,n-1 receive data. Mpi-process 0 will send to mpi-process n-1, mpi-process 1 will send to mpi-process n-2, and so on.

The **producer** program parameters are:

-l ***i*** ***i*** is the loop count.
-n ***j*** ***j*** is the number of bytes to transfer for each send operation.

As a first test, run **producer** between any pair of two nodes, nodeX and nodeY:

```
% mpimon producer -l 1 -n 1024 -- nodeX nodeY
```

A single mpi-process is started on each node, and a single message of size 1024 bytes are transferred from the mpi-process on nodeX to the mpi-process on nodeY. The program should return TEST COMPLETE.

Repeat the test for all pairs of nodes. N is the number of nodes (must be an even number for this test).

```
% mpimon producer -l 1 -n 1024 -- <node1> <node2>...<nodeN>
```

The program should return TEST COMPLETE.

2.6.2 Bandwidth - a bandwidth MPI test program

Bandwidth is a program to measure bandwidth for various message sizes between two mpi-processes. First one-way bandwidth and the latency for a zero byte message are measured, then the ping-pong (two-way) bandwidth and latency are measured.

Measure the bandwidth between any pair of nodes, nodeX and nodeY, by running:

```
% mpimon bandwidth -- nodeX nodeY
```

2.6.3 Bidirect - a bidirectional MPI test program.

Bidirect tests uni- and bi-directional traffic between a given number of nodes.

The program may be run between two nodes, nodeX and nodeY, using the **run_bidirect s**cript as:

```
% run_bidirect nodeX nodeY
```

or between a given set of nodes, nodeX nodeY... nodeZ, using another script as:

```
% run_permutated_bidirect nodeX nodeY... nodeZ
```

The **run_permutated_bidirect** script will test uni- and bi-directional traffic between all permutations of node combinations.

Chapter 3 Description of ScaMPI

3.1 General description

ScaMPI consists of libraries to be linked and loaded with the user application program(s) and a set of executables which control the start-up and execution of the user application program(s).

3.1.1 ScaMPI libraries

Name	Description
libmpi	Standard library containing the C API.
libfmpi	Library containing the Fortran API wrappers.

Table 3-1: Libraries

3.1.2 ScaMPI executables

A number of executable programs are included in ScaMPI.

3.1.2.1 mpimon - monitor program

mpimon is a monitor program which is the user's interface for running the application program.

3.1.2.2 mpisubmon - submonitor program

mpisubmon is a submonitor program which controls the execution of application programs. One submonitor program is started on each node per run.

3.1.2.3 mpiboot - bootstrap program

mpiboot is a bootstrap program used when running in manual-/debug-mode.

3.1.2.4 mpid - daemon program

mpid is a daemon program running on all nodes that can run ScaMPI. **mpid** is used for starting the **mpisubmon** programs (to avoid using Unix facilities like the remote shell **rsh**). **mpid** is started automatically when a node boots, and must run at all times.

3.2 Starting ScaMPI application programs

ScaMPI uses socket communication for control purposes. Schematically, start-up of application programs in a Scali System is performed as described in the following sections.

3.2.1 Application start-up - phase 1

- **Parameter control.**

mpimon does as much control of the specified options and parameters as possible. The userprogram names are checked for validity, and the nodes are, using sockets, contacted to ensure they are responding and that **mpid** is running.

- **Connecting to nodes.**

mpimon establishes a connection to the **mpid** daemon on each node specified, and transfers basic information to enable the daemon to start the submonitor **mpisubmon**.

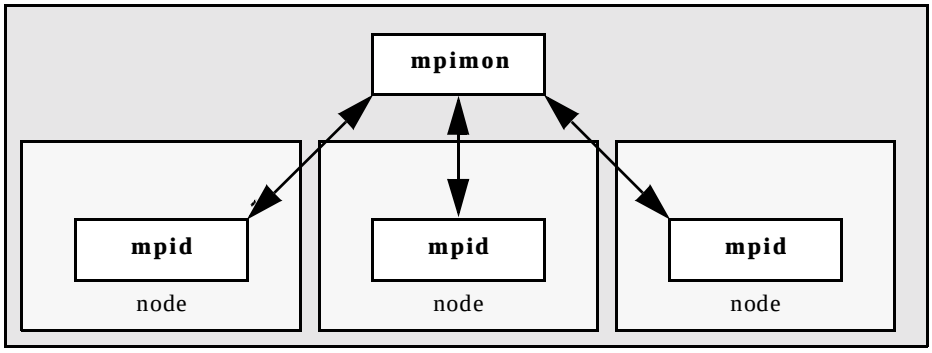

Figure 3-1: Application start-up - phase 1

3.2.2 Application start-up - phase 2

- **Starting submonitors.**

On each node, **mpid** starts the submonitor **mpisubmon**.

- **Transferring control information.**

Each submonitor establishes a connection to **mpimon.** Control information are exchanged between each **mpisubmon** and **mpimon** to enable **mpisubmon** to start the specified userprograms (mpi-processes).

- **Creating shared memory**

On each node, **mpisubmon** creates memory segments to be shared between the interconnected nodes.

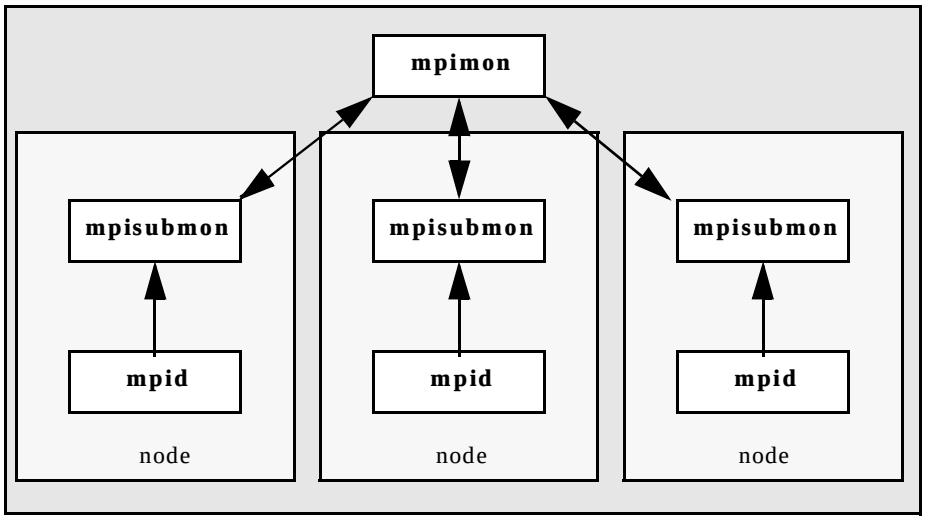

Figure 3-2: Application start-up - phase 2

3.2.3 Application start-up - phase 3

- **Starting mpi-processes.**

On each node, **mpisubmon** starts all the mpi-processes to be executed. Processes start and enter MPI_Init().

- **Mpi-processes synchronize.**

Upon receipt of all control information, the processes will via the local **mpisubmon** inform **mpimon** that they are ready to run. When all processes are ready, **mpimon** will return a 'start running' message to all the processes.

- **MPI-processes return from MPI_Init() and start to run.**

The user program(s) takes control.

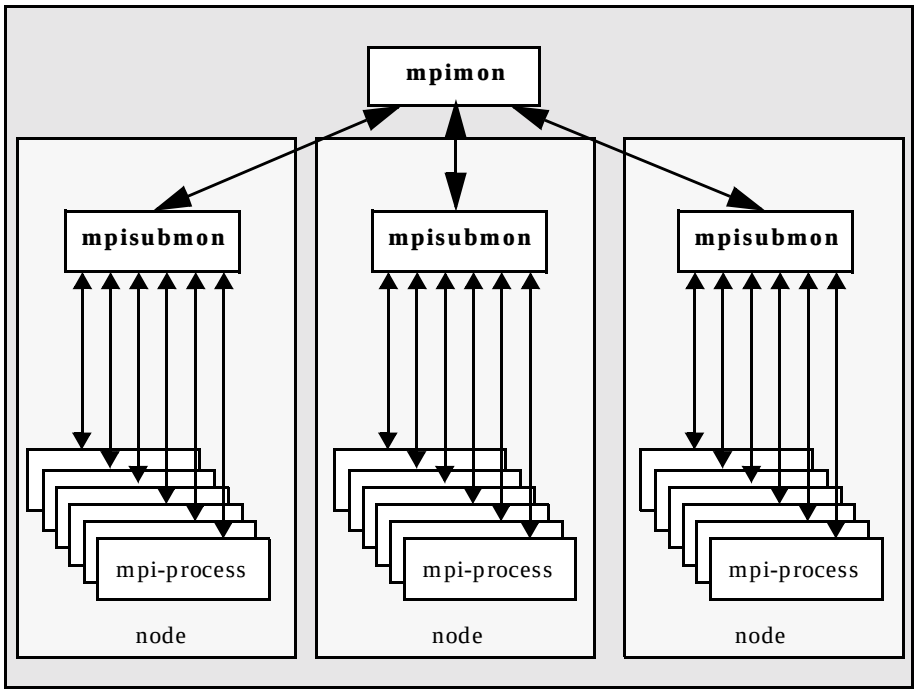

Figure 3-3: Application start-up - phase 3

3.3 Stopping ScaMPI application programs

Termination of application programs in a Scali System are performed as outlined below.

- **Mpi-processes enters MPI_Finalize().**

Each process signals, via its local **mbisubmon**, to **mpimon** that it has entered **MPI_Finalize**(), and it is now waiting.

- **Mpi-processes synchronize**

Processes wait for an "all stopped message" from **mpimon.** The message is transmitted via **mpisubmon** when all processes are waiting in **MPI_Finalize**().

- **Mpi-processes leave MPI_Finalize().**

Processes terminate, each **mpisubmon** releases shared memory segments and exits, and finally **mpimon** terminates.

3.4 Communication protocols

In ScaMPI, the communication protocol (*inlining*, *eagerbuffering*, *transporter*) used to transfer data between a sender and a receiver depends on the size of the message to transmit, see figure below.

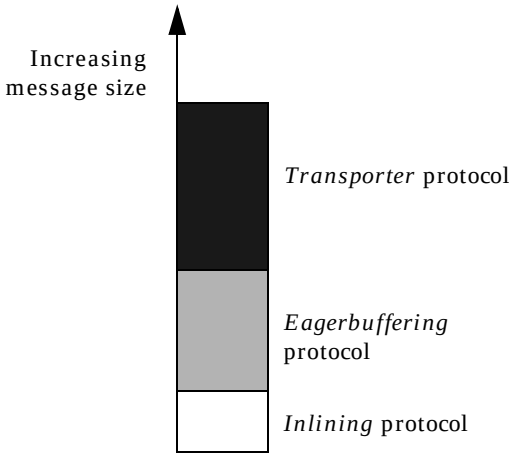

Figure 3-4: Thresholds for different communication protocols

The various communication protocols used, are briefly outlined in the following sections.

3.4.1 Inlining protocol

The *inlining* protocol is used when small messages are to be transferred.

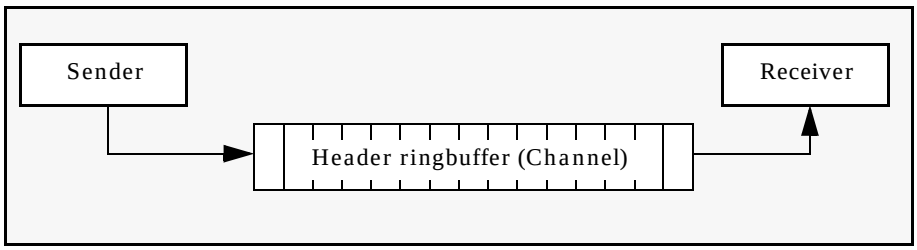

Figure 3-5: Inlining protocol

When the inlining protocol is used, the application's data is included in the message header. The inlining protocol utilizes one or more channel ringbuffer entries. The actual threshold for the inlining protocol can be set as described in section 3.5.1.

The inlining protocol is selected when:

0 <= message size <= channel_inline_threshold.

3.4.2 Eagerbuffering protocol

The *eagerbuffering* protocol is used when medium size messages are to be transferred.

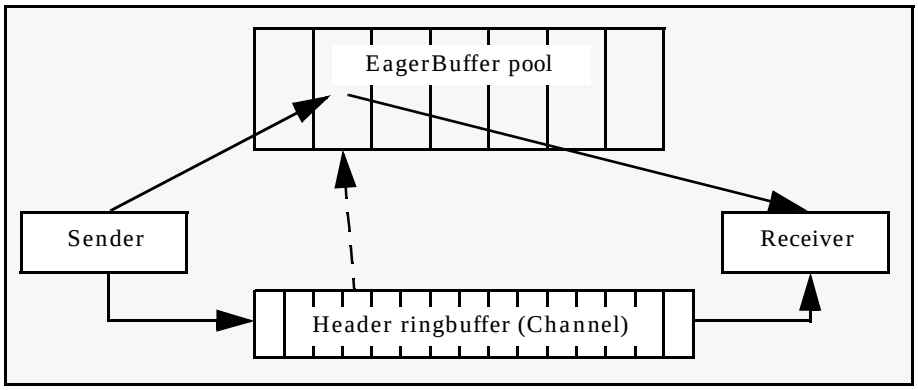

Figure 3-6: Eagerbuffering protocol

The protocol uses a scheme where the buffer resources, being allocated by the sender, are released by the receiver, without any explicit communication between the two communicating partners.

The eagerbuffering protocol utilizes one channel ringbuffer entry for the message header, and one eagerbuffer for the application data being sent.

The eagerbuffering protocol is selected when:

channel_inline_threshold < message size <= eager_size.

3.4.3 Transporter protocol

The *transporter* protocol is used when large messages are to be transferred.

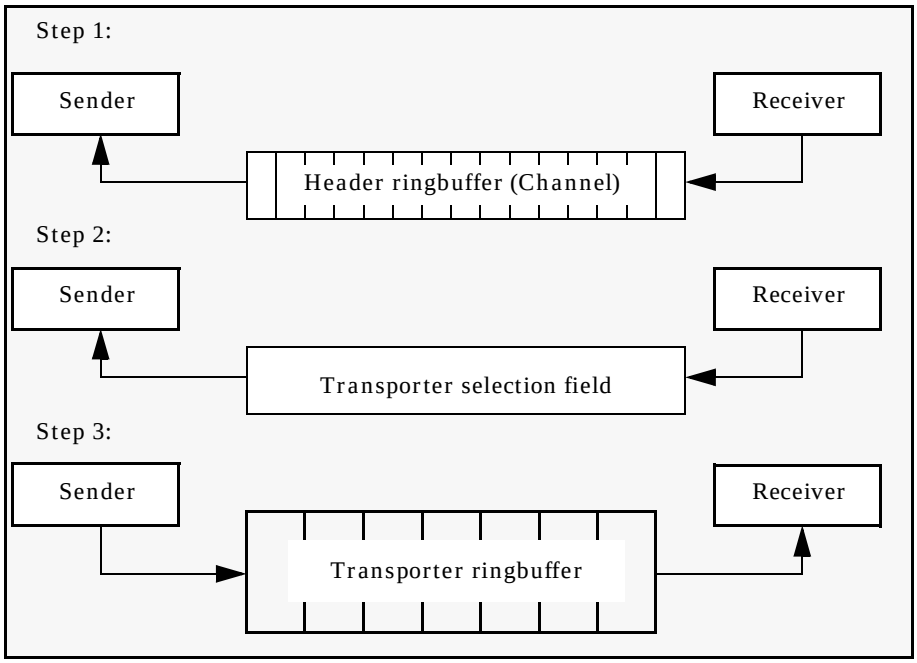

Figure 3-7: Transporter protocol

Initially (step 1), the protocol only transmits the message header. Once the receiver is ready to accept data (step 2), the sender is informed. Finally (step 3), the application's data is transferred from the sender to the recipient in the transporter ringbuffer.

The transporter protocol utilizes one channel ringbuffer entry for the message header, and transporter buffers for the application data being sent. The transporter protocol provides for fragmentation and reassembly of large messages, if necessary, for messages whose size is larger than the size of the transporter ringbuffer-entry (transporter_size).

The transporter protocol is selected when:

message size > eager_size.

3.5 Communication resources

All resources (buffers) used by ScaMPI reside in shared memory, and are allocated by **mpisubmon** on demand from the sender mpi-process. ScaMPI uses a *on demand* scheme for allocating resources. *On demand* means that buffers are not allocated until needed. To get a list of the resource settings, pass the **-verbose** option to **mpimon**.

mpisubmon operates on two separate buffer pools suitable for sharing - both pools in shared memory. One pool (local shared memory) provides resources for *intra-node* communication, and the other pool (SCI shared memory) provides resources for *inter-node* communication. The size of each buffer pool, and the size of each chunk may be set using options to **mpimon.** The *pool size* limits the total amount of shared memory, and the *chunk size* limits the maximum block of memory that can be allocated as a single buffer.
To set the *pool size* and the *chunk size* limits, use **mpimon** and specify:

-intra_pool_size <size>	to set the buffer pool size for intra-node communication
-intra_chunk_size <size>	to set the chunk size for intra-node communication
-inter_pool_size <size>	to set the buffer pool size for inter-node communication
-inter_chunk_size <size>	to set the chunk size for inter-node communication

Sections 3.5.1 - 3.5.3 outlines the various types of resources (*channel, eagerbuffer, transporter*) being used, and lists the **mpimon** options used to enforce specific buffering. However, it is normally **not** necessary to set the buffer parameters. Automatic buffer management is performed by ScaMPI, as described in the '*automatic buffer management*' section.

3.5.1 Channel buffer

For a sender-receiver pair, one *channel* ringbuffer is used for each communicator.

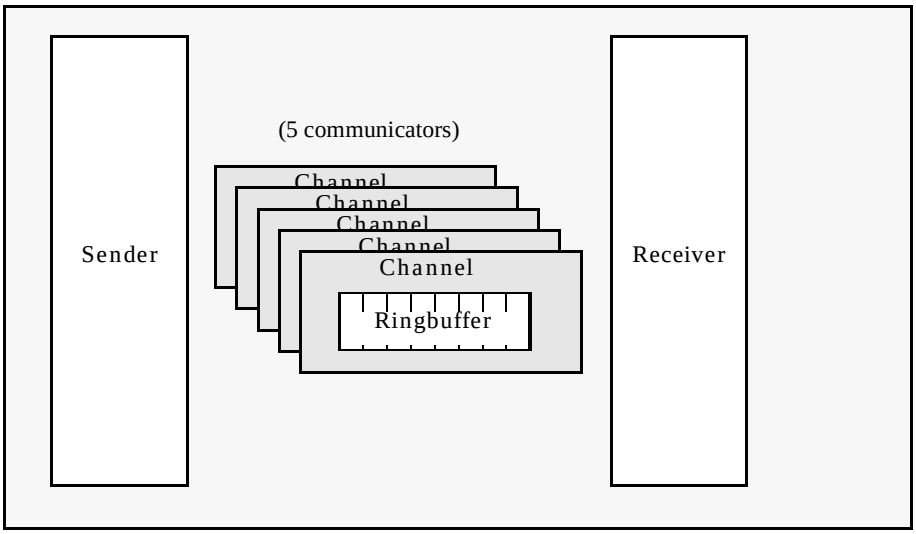

Figure 3-8: Channel resource

The ringbuffer is divided into equally sized entries. The size varies differs for different architectures/SCI-hardware; see "ScaMPI release notes" for details. An entry in the ringbuffer, which is used to hold the information forming the message envelope, is reserved each time a message is being sent, and is utilized by both the *inline* protocol, the *eagerbuffering* protocol, and the *transporter* protocol. In addition, one ore more entries are utilized by the *inline* protocol for application data being transmitted.

To force the *channel* resource definitions, use **mpimon** and specify:

-intra_channel_size <size>	to set the ringbuffer size (in bytes) per intra-channel
-inter_channel_size <size>	to set the ringbuffer size (in bytes) per inter-channel

To set the *channel* threshold definitions, use **mpimon** and specify:

-intra_channel_inline_threshold <size>	to set threshold for inlining per intra-channel
-inter_channel_inline_threshold <size>	to set threshold for inlining per inter-channel

3.5.2 Eagerbuffer buffer

For a sender-receiver pair, one, and only one, *eagerbuffer* buffer is used.

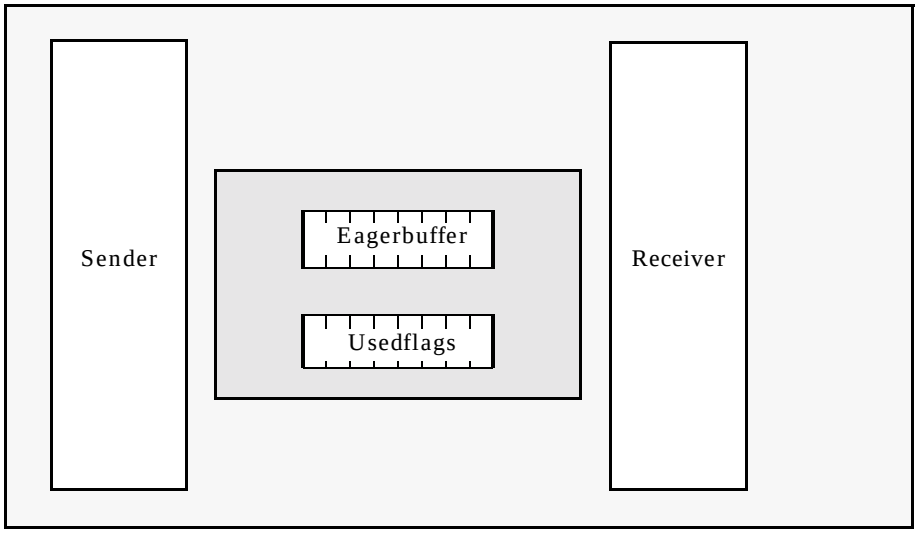

Figure 3-9: Eagerbuffer buffer

An *eagerbuffer* buffer is allocated when medium size messages are to be transferred, and is utilized by the *eagerbuffering* protocol.

To change the *eagerbuffer* resource definitions, use **mpimon** and specify:

-intra_eager_size <size>	to set the buffer size (in bytes) for intra-node communication
-intra_eager_count <count>	to set number of buffers for intra-node communication
-inter_eager_size <size>	to set the buffer size (in bytes) for inter-node communication
-inter_eager_count <count>	to set number of buffers for inter-node communication

3.5.3 Transporter buffer

For a sender-receiver pair, one, and only one, *transporter* buffer set is used.

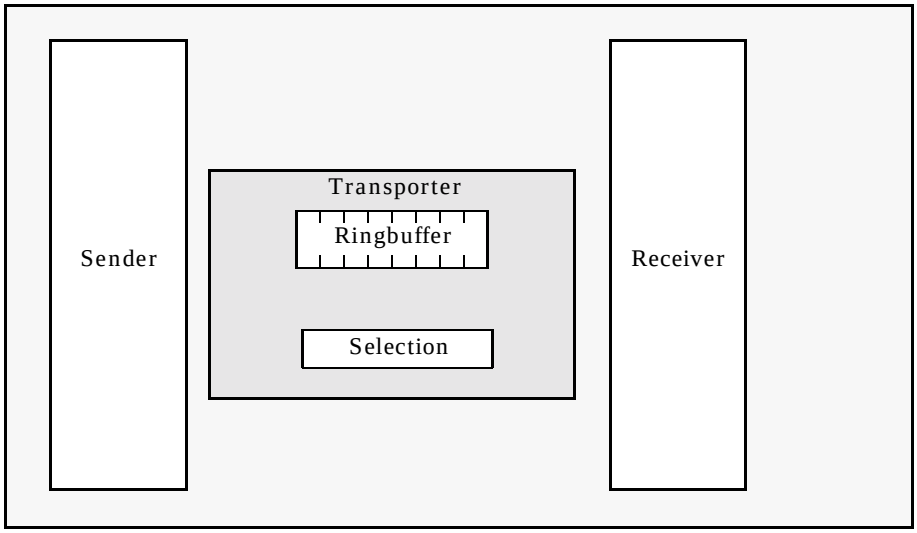

Figure 3-10: Transporter buffer

A *transporter* buffer is allocated when large messages are to be transferred, and is utilized by the *transporter* protocol. The buffer consists of equally sized entries arranged as a ringbuffer.

To change the *transporter* resource definitions, use **mpimon** and specify:

-intra_transporter_size <size>	to set the bufferentry size (in bytes) for intra-node communication
-intra_transporter_count <count>	to set number of entries in buffer for intra-node communication
-inter_transporter_size <size>	to set the bufferentry size (in bytes) for inter-node communication
-inter_transporter_count <count>	to set number of entries in buffer for inter-node communication

Chapter 4 Tips & Tricks for ScaMPI

This chapter is the place to start when something seems to go wrong running your ScaMPI programs. If you have any problems with ScaMPI, first check the (not yet complete) list of common errors and their solutions. An updated list of ScaMPI Frequently Asked Questions are posted in the *Support* section at **http://www.scali.com**. If you cannot find a solution to the problem(s), please read this chapter before contacting **support@scali.com**.

Currently, the following sections are by no means complete. Problems reported to Scali will eventually be included in appropriate sections. Thus, please send your relevant remarks by e-mail to **support@scali.com**.

4.1 Application program notes

4.1.1 MPI_Probe() and MPI_Recv()

During development and test of ScaMPI, we have run into several application programs with the following code sequence:

```
while (...) {
      MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, sts);
      if (sts->MPI_TAG == SOME_VALUE) {
            MPI_Recv(buf, cnt, dtype, MPI_ANY_SOURCE,
                  MPI_ANY_TAG, comm, sts);
            doStuff();
      }
      doOtherStuff();
}
```

For MPI implementations that have one, and only one, receive-queue for all senders, the program's code sequence works ok. However, the code will **not** work as expected with ScaMPI. ScaMPI utilizes one receive-queue per sender (inside each mpi-process). Thus, a message from one sender can bypass the message from another sender. In the time-gap between the completion of **MPI_Probe()** and before **MPI_Recv()** matches a message, another new message from a different mpi-process could arrive, i.e., it is not certain that the message found by **MPI_Probe()** is identical to one that **MPI_Recv()** matches.

To make the program work as expected, the code sequence should be corrected to:

```
while (...) {
       MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, sts);
       if (sts->MPI_TAG == SOME_VALUE) {
              MPI_Recv(buf, cnt, dtype, sts->MPI_SOURCE,
                     sts->MPI_TAG, comm, sts);
              doStuff();
       }
       doOtherStuff();
}
```

4.1.2 Unsafe MPI programs

Because of different buffering behaviour, some programs may run with MPICH, but **not** with ScaMPI. Unsafe MPI programs may require resources that are not always guaranteed by ScaMPI, and deadlock might occur (since ScaMPI use spinlocks, these might seem to be livelocks). If you want to know more about how to write portable MPI programs, see for example [2].

A typical example that will **not** work with ScaMPI (for long messages):

```
while (...) {
       MPI_Send(buf, cnt, dtype, partner, tag, comm);
       MPI_Recv(buf, cnt, dtype, MPI_ANY_SOURCE,
              MPI_ANY_TAG, comm, sts);
       doStuff();
}
```

To get this example to work with ScaMPI, the **MPI_Send()** must either be replaced by using **MPI_Isend()** and **MPI_Wait()**, or the whole construction should be replaced using **MPI_Sendrecv()** or **MPI_Sendrecv_replace()**.

4.2 Namespace pollution

The ScaMPI library, being written in C++, have all its class names prefixed with **MPI_**. Depending on the compiler used, the user may run into problems if he/she has C++ code using the same prefix **MPI_**. In addition, there exist a few global variables that could cause problems. All these functions and variables are listed in the include files **mpi.h** and **mpif.h**. Normally, these files are installed in /**opt/scali/include**.

Due to the fact that ScaMPI doesn't have fixed its OS routines to specific libraries, it will be good programming practise to avoid using OS functions as application function names. Naming routines or global variables as **send**, **recv**, **open**, **close**, **yield**, **internal_error**, **failure**, **service** or other OS reserved names may result in an unpredictable and undesirable behaviour.

4.3 Error and warning messages

4.3.1 User interface errors and warnings

User interface errors are problems with the environment setup causing difficulties for **mpimon** when starting a ScaMPI program. **mpimon** will not start before the environment is properly defined. These problems are usually easy to fix, by giving **mpimon** the correct location of some executable. The error message provides a straight forward indication of what to do. Thus, only particularly troublesome user interface errors will be listed here.

Using the **-verbose** option enables **mpimon** to print more warnings.

4.3.2 Fatal errors

Upon a fatal error, ScaMPI prints an error message before calling **MPI_Abort()** to shut down all mpi-processes.

4.4 When things don't work - troubleshooting

This section is meant as a starting point to help debugging. The main focus is on locating and repairing faulty hardware and software setup, but can also be helpful in getting started after installing a new system. For a description of the Scali Universe GUI, see the Scali System Guide [4].

4.4.1 Standard input and ScaMPI

The **-stdin** option specifies which mpi-process rank should receive the input. You can in fact send **stdin** to all the mpi-processes with the **all** argument, but this requires that all mpi-processes read the exact same amount of input. The most common way of doing it is to send all data on **stdin** to rank 0:
mpimon **-stdin 0** myprogram -- node1 node2 ... < input_file
Note that default for -**stdin** is **none**.

4.4.2 Why doesn't my program start to run?

? mpimon: command not found.
☺ Include /opt/scali/bin in the PATH environment variable.

? mpimon can't find mpisubmon.
☺ Set MPI_HOME=/opt/scali or use the -execpath option.

? The application has problems loading libraries (libsca*).
☺ Update the LD_LIBRARY_PATH to include /opt/scali/lib.

? Incompatible mpi versions.
mpid, mpimon, mpisubmon and the libraries all have version variables that are checked at start-up.
☺ Set the environment variable **MPI_HOME** correctly
☺ Restart mpid because a new version of ScaMPI is installed without restart of **mpid**
☺ Reinstall ScaMPI because a new version of ScaMPI was not cleanly installed on all nodes.

? Set workingdirectory failed
☺ ScaMPI assumes homogenous file-structure. If you start mpimon from a directory that is not available on all nodes you must set SCAMPI_WORKING_DIRECTORY to point to a directory that are available on all nodes.

? ScaMPI uses wrong interface for tcp-ip on frontend with more than one interface
☺ Set SCAMPI_NODENAME to hostname of correct interface.

? MPI_Wtime gives strange values
☺ ScaMPI uses a hardware-supported highprecision timer for MPI_Wtime. This timer can be disabled by SCAMPI_DISABLE_HPT=1

4.4.3 Why doesn't mpid start

mpid opens a socket and assigns a predefined mpid port number, see /etc/services, to the end point. If mpid is terminated abnormally, the mpid port number cannot be re-used until a system defined timer has expired.
☺ Use **netstat -a | grep mpid** to observe when the socket is released. When the socket is released, restart **mpid** again.

4.4.4 Interconnect problems

4.4.4.1 Routing

? The program terminates with an ICMS_NO_RESPONSE error message
This happens when one or more mpi-processes are unable to create a remote memory mapping to another node within a (long) period of time.
☺ Check if all relevant nodes are alive by issuing any command with scash, e.g., /opt/scali/bin/scash -p nodename.

☺ Check if SCI network routing is properly set with **/opt/scali/sbin/scaconftool** (command: **sciping OK**), or use the Scali Universe GUI.

4.4.4.2 Bad clean up

? A previous ScaMPI run has not terminated properly.
☺ Check for mpi-processes on the nodes using **/opt/scali/bin/scaps**.
☺ Use **/opt/scali/sbin/scidle**
☺ Use **/opt/scali/bin/scash** to check for leftover shared memory segments on all nodes (**ipcs** for Solaris and Linux).

Note that core dumping takes time...

4.4.4.3 Space overflow

? The application have required too much SCI or shared memory resources.
☺ Your **mpimon** *pool-size* specifications are too large.
☺ Number of communicators in the program is higher than expected when doing automatic buffer calculations. Since memory by default is allocated in large chunks, try to reduce the *chunk-size* parameter to **mpimon**
(use **mpimon -verbose** to get current buffer settings).

4.4.5 Why does my program terminate abnormally?

4.4.5.1 Core dump

? The application core dumps.
☺ Use a debugger to locate the point of violation. The application may need to be recompiled to include symbolic debug information (**-g** for most compilers).
☺ Define SCAMPI_INSTALL_SIGSEGV_HANDLER=1 and attach to the failing process with debugger.

4.4.5.2 SCI interconnect failures

? The program terminates with an ICMS_* message
☺ An SCI problem has occurred, find out more using the SCI diagnostics helper: /opt/scali/bin/sciemsg <error-code>. Reloading of SCI drivers and rerouting your system may be necessary. Contact your local System Administrator if assistance is needed. The interconnect diagnostic in the Scali Universe GUI and the SCI documentation in the Scali System Guide may help you locate the problem. Problems and fixes will be included in the FAQ on **http://www.scali.com**. If there is a SCI problem needing attention, please contact **support@scali.com**.

4.4.5.3 General problems

? Are you reasonable certain that your algorithms are MPI safe?
☺ Check if every send has a matching receive.

? The program just hangs

☺ Try starting the program with **-init_comm_world** specified;
if it doesn't start, there is a buffer allocation problem. Further information is available in the '*How do I control SCI and local shared memory usage?*' section.

☺ If the application has a large degree of asynchronicity, try to increase the *channel-size*. Further information is available in the '*How do I control SCI and local shared memory usage?*' section. Are you really sure that your algorithms are MPI safe?

? The program terminates without an error message

☺ Investigate the core file, or rerun the program in a debugger.

4.4.6 How do I control SCI and local shared memory usage?

? Adjusting ScaMPI buffer sizes

Note that forcing size parameters to **mpimon** is usually **not** necessary. This is only a means of optimising ScaMPI to a particular application, based on knowledge of communication patterns. For unsafe MPI programs it may be required to adjust buffering to allow the program to complete.

? How do I control SCI and local shared memory usage?

The **eager buffer**s are used for small messages, while the **transporter buffer**s are used for handling large messages (larger than eager size).
The **channel buffer**s is a send queue where each entry is 64 bytes, i.e.in a 8k buffer there is room for 128 outstanding requests. The function of the various buffers is outlined in section 3.5. All buffers are created when needed (i.e., when tried used for the first time), or at start up when **-init_comm_world** is specified.
The buffer space required by a communication channel is approximately:

channel = (2 * channel-size * communicators)
+ (transporter-size * transporter-count)
+ (eager-size * eager-count)
+ 512 (give-or-take-a-few-bytes)

Note: Messages up to 560 bytes (the upper limit can be set using the option **channel_inline_threshold <size>** to **mpimon**) get inlined in the channel buffer. If you frequently use short messages, increasing the *channel-size* beyond 4k bytes might be a good idea.

4.4.6.1 Automatic buffer management

The communicators parameter depends on the application (assumed to be two in the automatic approach). If more communicators than expected by the buffer size calculations are used, the application may run out of shared memory. To overcome this, reduce the *chunk-size*.
The *pool-size* is a limit for the total amount of shared memory.

The automatic buffer size computations is based on a full connectivity, i.e., all communicating with all others. If all mpi-process *P* in a program communicate with all the other mpi-processes, each mpi-process will communicate with *P_intra* mpi-processes intra node (it-self inclusive) and *(P - P_intra)* mpi-processes inter node. Given a total *pool* of memory dedicated to communication, each communication channel will be restricted to use a partition of only:
inter_partition = inter_pool_size / (P_intra*(P-P_intra))
intra_partition = intra_pool_size / (P_intra * P_intra)
The automatic approach is to downsize all buffers associated with a communication channel until it fits in its part of the pool. The *chunk size* sets the size of each individual allocated memory segment. The automatic chunk size is calculated to wrap a complete communication channel.

4.5 How to optimize MPI performance

There is no universal recipe for getting good performance out of a message passing program. Here are some do's and don't's for ScaMPI.

4.5.1 Performance analysis.

Learn about the performance behaviour of your MPI application on a Scali System by using a performance analysis tool.
The freely available ScaMPE profiling library may be used with ScaMPI. For more information, please see section 2.4.3.

4.5.2 Using MPI_Isend(), MPI_Irecv().

If communication and calculations does not overlap, using immediate calls, e.g., **MPI_Isend()** and **MPI_Irecv()**, are usually performance ineffective.

4.5.3 Using MPI_Bsend().

Using buffered send, e.g., **MPI_Bsend()**, usually degrade performance significantly compared to their unbuffered relatives.

4.5.4 Avoid starving mpi-processes - fairness.

MPI programs may, if not special care is taken, be unfair and may starve mpi-processes, e.g., by using **MPI_Waitany()** as illustrated for a client-server application in example 3.15 & 3.16 in the MPI 1.1 standard [1]. Fairness can be enforced, e.g., by use of several tags or separate communicators.

4.5.5 Using processor-power to poll.

ScaMPI is implemented using poll when waiting for communication to terminate. This is efficient when this period is short or if you don't have anything else to use the processorpower for. In threaded application with irregular communication patterns you probably have other threads that could make use of the processor. In this case performance may increase if you enable the backoff-polling-strategy built into ScaMPI. It functions like this: After waiting a short period (idle time) we start backing off using systemcall nanosleep to release processor. The nanosleep period starts at a minimum and it doubles for each call until it reaches a maximum. It is controlled by a set of environment-variables:

SCAMPI_BACKOFF_ENABLE	turns the mechanism on
SCAMPI_BACKOFF_IDLE	=n defines idle-period to n ms Default 20 ms
SCAMPI_BACKOFF_MIN	= n defines minimum backoff-time in ms Default 10 ms
SCAMPI_BACKOFF_MAX	= n defines maximum backoff-time in ms Default 100 ms

4.5.6 Communication buffer adaption

If the communication behaviour of the application is known, explicitly giving buffersize settings to mpimon to match the requirement of the application, will in most cases improve performance.

Example: Application sending only 900 bytes messages.

☺ Set **channel-inline-threshold 964** (64 added for alignment) and increase the channel-size significantly (32-128 k).

Note: the channel-inline-threshold can not be increased beyond 1023.

☺ Setting **eager-size 1k** and **eager-count** high (16 or more).

Note: If all messages can be buffered, the transporter-{size, count} can be set to low values to reduce shared memory consumption.

4.5.7 Reorder network traffic to avoid conflicts

Many-to-one communication may introduce bottlenecks.

Zero byte messages are low-cost. In a many-to-one communication, performance may improve if the receiver sends ready-to-receive tokens (in the shape of a zero-byte message) to the mpi-process wanting to send data.

4.6 Benchmarking

Benchmarking is that part of performance evaluation that deals with the measurement and analysis of computer performance using various kinds of test programs. Benchmark figures should always be handled with special care when compared to similar results.

4.6.1 How to get expected performance

- **Improving performance for short runs.**
  By default, communication buffers are allocated when requested the first time. To eliminate this startup time from your measurement either run a warm-up phase before doing the actual measurement or use the parameter **-init_comm_world** to **mpimon** to allocate communication buffers between all pairs of mpi-processes.
- **Caching the application program on the nodes.**
  *F*or benchmarks with short execution time, total execution time may be reduced when running it repetitive. For large configurations, copying the application to the local file system on each node will reduce startup latency and improve disc bandwidth.
- **The first iteration is (very) slow.**
  The mpi-processes in an application are not started simultaneously. Inserting an **MPI_Barrier()** before the timing loop will eliminate this. To reduce setup time after **MPI_Init()**, specify the parameter **-init_comm_world** to **mpimon**.

4.6.2 Memory consumption increase after warm-up

Remember that group operations (**MPI_Comm**_{**create, dup, split***}*) may involve creating new communication buffers. If this is a problem, decrease the *chunk-size* as described in section 4.4.

Chapter 5 ScaShmem

Scali's compatibility library for Cray/SGI ShMem.

5.1 Description

The Scali's compatibility library covers most of the Cray/SGI ShMem application programmers interface (exceptions are listed in release-notes). The library is made to enable running of applications previously limited to Cray/SGI machines in a workstation environment. With the favourable price and availability of memory for workstations, memory intensive applications may in particular benefit from this library.

This implementation of Scali´s Cray/SGI ShMem compatibility library is layered on top of ScaMPI imposing some restrictions on the usage. The ShMem communication layer uses ScaMPI's thread-hot and -safe feature and creates an additional server thread to handle remote requests, i.e. a client- server architecture. For new applications it is therefore recommended to use ScaMPI as your communication library to utilize the full performance potential of Scali products.

5.2 Application porting to ScaShmem

Using of the Shmem compatibility library means running on another architecture than the Cray/SGI machine which your application initially was made for. The size of data types on your Cray/SGI system and your cluster may differ; e.g. processors employed in the T3E are true 64 bit processor while x86 PCs are 32 bit, and changes may have to be made to the source code before it can run on your new architecture. Make sure that your ShMem code is tolerant to these differences.

Some examples of do & don't:

- the T3E **<int>** is 64 bit, all such variables must be replaced with **<long long>** on a 32 bit machine to operate with the same precision
- make consistent use of header files (shmem.h vs. shmem.fh)
- use portable maths when comparing with T3E (i.e. -Kieee -pc 64)
- enable flags related to the T3E implementation as the source code may support other architectures as well.
- Fortran code with careful declaration of variables' size (kind=4 etc according to Cray compiler practice) and corresponding naming of the shmem procedure calls simplifies the porting process

- Zero initialization is not supplied by all Fortran and C compilers for workstations.

5.3 Features and limitations

Please refer to the ScaShMem release notes (**/opt/scali/doc/ScaShMem/RELEASE_NOTES**) for further details which may not have made it into this manual.

5.3.1 Communication initialization and termination

Since Cray/SGI ShMem does not have explicit start and stop function calls, the MPI and the server thread is started when the first call to a **shmem_*** function is performed.

5.3.1.1 Communication initialization

If your application has processes that rely on remote data before the corresponding process have performed a communication call, and the corresponding process have performed a communication call, you will have to either call the nonstandard function **shmem_start()** or do a **shmem_barrier_all()**.

5.3.1.2 Communication termination

All application processes are stopped when the first process terminates. It is therefore recommended to end your application with a **shmem_barrier_all()**. Your application will terminate with messages aka

```
-- mpimon --- Aborting run after process-<n> terminated abnormally
Childprocess <m> exited with exitcode 216 ---
```

5.3.2 Runtime requirements

Since we impose a client/server architecture we recommend having two processors per shmem process, i.e. one process per dual processor workstation. Running on single processor workstations will have serious negative impact on performance.

5.3.3 Datatypes / porting

The default data types size for put and get are 64 bit. Beware that integers for all architectures, and long's for x86, are 32 bit, or even better, use the shmem calls with specified size of data type.

5.3.4 Dynamic memory allocation

Unlike on a machine from SGI Inc. or Cray Inc., using standard Unix memory allocation to memory used in ShMem communication will not work. Dynamic memory visible to ShMem communication have to be allocated with one of the following:

```
void *shmalloc(size_t size);
void *shrealloc(void *ptr, size_t size);
void *shmemalign(size_t alignment, size_t size);

POINTER (addr, A(1))
INTEGER (length, errcode, abort)
CALL shpalloc(addr, length, errcode, abort);
```

Freeing ShMem capable memory must be done with:

```
void shfree(void *ptr);
```

No stack manipulation issues are implemented.

5.3.5 ScaShmem environment variables

The processes are allocated statically at startup. Standard Cray/SGI ShMem environment variables to control the placement of the ShMem application on the system are not recognized.

5.4 Compiling and linking

Fortran (be careful with your data size and initialization):

```
g77 -D_REENTRANT -I/opt/scali/include -c <options> appl.f
g77 -o appl appl.o -L/opt/scali/lib -lshmem -lfmpi -lmpi -lpthreads
```

C:

```
gcc -D_REENTRANT -I/opt/scali/include -c <options> appl.c
gcc -o appl appl.o -L/opt/scali/lib -lshmem -lfmpi -lmpi -lpthreads
```

Note: Libraries shall always come *after* the object files on the linkage line.

5.5 Running your application

To run your application you may use shmemrun like this

```
/opt/scali/bin/shmemrun -np <number of nodes> -coherence
<lazy/eager/automatic> appl <options>
```

For details run shmemrun -h.

If you prefer to use mpirun or mpimon please use the -shmem mpimon option (ScaMPI 1.11.9 or newer).

Chapter 6 ScaIP - IP for SCI

This chapter describes the Scali Internet Protocol driver **ScaIP**.

Please note that the **ScaIP** *release notes* and other readable information are available in the **/opt/scali/doc/ScaIP** directory.

6.1 Introduction

The ScaIP package contains the kernel mode driver **scip**, which when it is loaded, exists as a kernel-resident Internet Protocol network interface (supporting the 'inet' address family).

When properly configured, the scip driver provides support for the upper layered module Linux IP to transmit and receive Internet datagram packets over SCI.

6.2 Simplified network model

The network architecture spans both the user-level and the kernel-level, as shown in "Figure 6-1: Simplified network model". The application layer is executed at the user-level, and is utilizing the network services provided at the transport layer in the kernel. The most common predefined interface between the transport layer and the application layer is the Berkely socket interface. The transport layer protocols (e.g. TCP and UDP) use services offered by the network layer (IP) to send messages to a destination node and to receive messages from other nodes. The network layer handles the transfer of data packets between the connected nodes using the services provided by the link layer software (e.g. the Ethernet driver or the ScaIP/ScaMAC/ScaSCI modules). The link layer includes the network interface hardware and the software device driver software that controls the network interface.

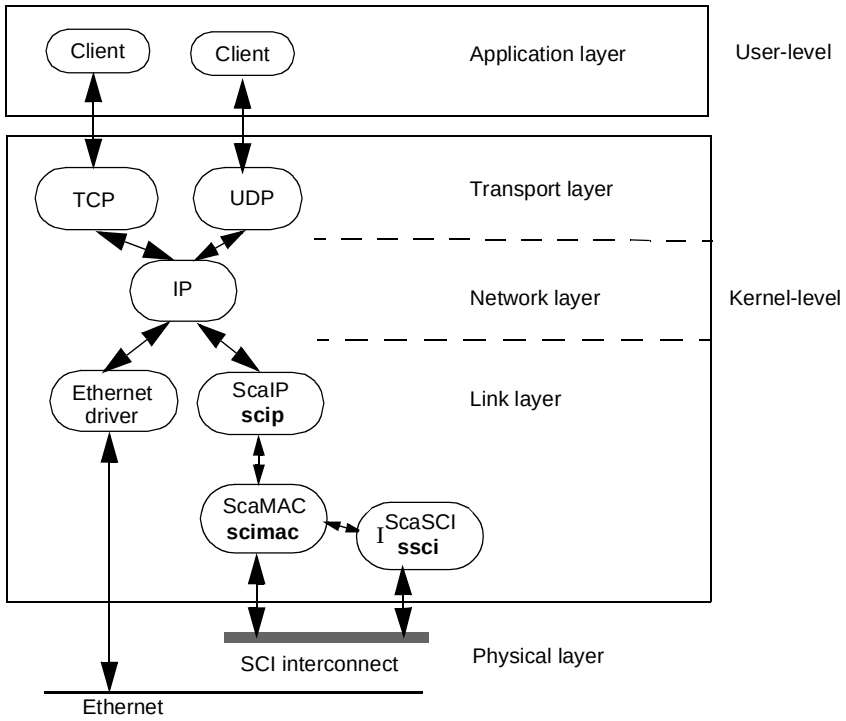

Figure 6-1: Simplified network model

6.3 Configuration

The ScaIP network interface driver **scip** is configured using the standard Linux maintenance command **/sbin/ifconfig**, or any other suitable Linux command or script. To display information about the ScaIP interface, the standard Linux command **/bin/netstat** can be used. Thus, standard Linux utilities are to be used both to configure ScaIP and to retrieve network interface information from the **scip** driver.

The name of the ScaIP interface is the driver name (**scip**) followed by the unit number (0 for the first ScaIP interface), for example **scip0**.

The address being assigned to the ScaIP interface (*inet* family) is an IP address (e.g. in a class-C network, 192.168.4.1 where 192.168.4 is the network number and 1 is the node's unique host number).

By default, the Address Resolution Protocol (ARP) is enabled to implement mapping between the IP address and the hardware address for the network interface. The use of the ARP protocol can be disabled. To manually create the address mapping, the Linux maintenance command **`/sbin/arp`** can be used. If, for one reason or another, there is a need to enforce a statically (and permanently) creation of address mapping entries for nodes accessible via the ScaIP interface, consult the text file '/opt/scali/kernel/scip/arptable.example' and follow the instructions contained therein.

The hardware address of the ScaIP interface is associated with the interface when the scip driver is loaded and attached to the ScaMAC module (i.e., the scimac driver). It is not permitted to change the hardware address of the interface.

The scip driver does not provide fragmentation of datagram packets. If transmission of a datagram larger than the Maximum Transmission Unit (MTU) is attempted, the datagram packet is dropped. The physical MTU size of the ScaIP interface cannot be set using '/sbin/ifconfig' (or any other utility program). The MTU setting is determined by a configuration property (scimac_max_ebuf_size) for the ScaMAC module; the actual MTU of the ScaIP interface is 'scimac_max_ebuf_size + 86'.

The scip driver is initialized when the ScaIP module is loaded into Linux kernel space, and is de-initialized when the ScaIP module is unloaded from the Linux kernel.

Error messages from the driver are printed to the system's log file (e.g. **`/var/log/messages`**).

6.4 ScaIP package installation

The **scip** driver cannot be used to transfer packets over SCI until the ScaIP package (distributed as a Linux RPM file) has been installed on each processing node.

To install the ScaIP package you *should* use the Scali Software Platform (SSP) installation program. The SSP installation program provides you with the option to install the ScaIP package on each of the selected processing nodes.

The **scip** driver depends on the **scimac** driver (the Scali Media Access Control driver for SCI) included in the ScaMAC package, which in turn depends on the **scasci** driver (the Scali PCI SCI driver) included in the ScaSCI package.

The chapter *Software Installation* in the Scali System Guide offers detailed information on how to use the SSP installation program.

Another way to install (or update) the ScaIP package is interactively by hand using the Linux RPM package manager program **rpm**. Note that the operation should be performed on each of the processing nodes.

For a description of how to update the ScaIP package using the **rpm** program, please consult the text file **/opt/scali/doc/ScaIP/INSTALL.**

Chapter 7 ScaMAC

This chapter describes the Scali Media Access Control package ScaMAC.

7.1 Introduction

The ScaMAC package has been developed to provide an efficient way to pass data packets between SCI interconnected nodes.

The package includes the kernel mode driver **scimac** and some utility programs. The **scimac** driver is written as a multi-threaded loadable driver module supporting unicast data transfer over the SCI interconnect.

In the current implementation neither broadcasting to all nodes nor multicasting to a group of nodes on the interconnect are supported.

Please note that the ScaMAC *release notes* and other readable information are available in the **/opt/scali/doc/ScaMAC** directory.

For a brief description of the network model, see chapter 6.2 on page 57.

7.2 The scimac driver

The **scimac** driver provides Application Programming Interface services for other upper layered modules (e.g. ScaIP) to transmit and receive data packets over SCI. The API contains methods for other modules to attach/detach to **scimac**, and methods to exchange data packets with similar modules on other SCI interconnected nodes.

The **scimac** drivers provide a reliable connection oriented (node-to-node) transfer of data packets over SCI, and guarantees that the data is sent and received in order. Upon SCI interconnect problems, data packets are, by default, retransmitted until retransmission timeout. On timeout, the data packet is dropped by **scimac**.

Upon error, **scimac** error messages and warnings are printed to the system log file (e.g. **/var/log/messages**).

7.3 Setting up the scimac driver

The **scimac** driver can be explicitly started, stopped and restarted by user *root* using the command:

```
# /sbin/service/scimac [start|stop|restart]
```

Note that the **scimac** driver depends on the **ssci** driver (Scali PCI SCI driver) to be running. If the Scali PCI SCI driver is not loaded and started, the **scimac** driver will fail to start.

The file **/opt/scali/kernel/scimac/scimac.conf** contains configuration variables for the **scimac** driver. If a variable is changed, the new value will override the default value when the **scimac** module is installed in the kernel. Note that a modification to **scimac.conf** will not be effective until the **scimac** driver module is reloaded.
Table 7-1 lists the configuration variables together with their default values.

Variable	Default	Description
scimac_max_no_hdrs	32	The maximal number of **scimac** packet headers to be used by the driver
scimac_max_no_ebufs	8	The maximal number of *eager* buffers to be used by the **scimac** driver
scimac_max_ebuf_size	32768	The maximal size in bytes of each *eager* buffer used
scimac_use_ulevel_recv	1	Enable use of a kernel thread to defer reception of packets
scimac_use_sw_interrupts	0	Enable use of the immediate bottom half to defer reception of packets
scimac_max_send_queuelen	2000	The maximal number of packets queued for transfer per connection path at any one time
scimac_pkt_rexmit_time	200	The packet retransmit time in milliseconds
scimac_max_rexmit_time	5000	The packet's maximal retransmit time in milliseconds

Table 7-1: scimac configuration variables

Variable	Default	Description
scimac_min_nodeid_number	0x100	The lowest valid node identifier in the Scali system
scimac_max_nodeid_number	0xff00	The largest valid node identifier in the Scali system
scimac_nodeid_increment	0x100	The incremental node identifier step in the Scali system

Table 7-1: scimac configuration variables

Normally, there should not be necessary to change any of the configuration parameters.However, the configuration is by default optimized for a Linux 2.4.x kernel. If unexpected performance problems occur when using a Linux 2.2.x kernel, you should change two of the default configuration values. For a Linux 2.2.x kernel, edit the file: **`/opt/scali/kernel/scimac/scimac.conf`** by setting:

```
scimac_max_ebuf_size    = 16384
```

7.4 The ScaMAC utilities

The following is a description of the ScaMac utility programs which can be used to retrieve and display various **scimac** driver interface information, and to check the reachability of remote **scimac** drivers. The utilities are located in the **bin** and the **sbin** directories of the Scali installation. Normally they are of limited interest to the ordinary user. Currently, no standard Linux utilities can be used to collect and display information about the **scimac** driver interface.

7.4.1 macstat - display scimac driver status

The program **macstat** displays information gathered from the driver's data structures. The information printed is controlled by the option you select.

Usage:

```
/opt/scali/bin/macstat -{a|c|i|s|R|p} [<ppa number>]
```

Options:

`-a` Display the state of all remote node connections, and connection configuration related information.

-c	Show the **scimac** driver's configuration variables (current values), as defined by the configuration properties listed in the text file **/opt/scali/kernel/scimac/scimac.conf.**
-i	Display the state of all remote node connections, and connection traffic statistics.
-s	Show a summary of accumulated driver traffic.
-R	Reset the traffic statistics (privileged, for user *root* only).
-p	Show the distribution of data packet sizes being transferred.
<ppa number>	Access the specified scimac instance (physical point of attachment), instead of 0 (default).

7.4.2 macping - check reachability of remote scimac drivers

The program **macping** can be used to check the reachability and connectivity of **scimac** drivers on SCI interconnected nodes. When the program is activated, the **scimac** driver sends a short out-of-band request packet via the SCI interconnect to the **scimac** driver(s) on the nodes specified at the command line. If a node responds, **macping** computes the round-trip time and prints a summary of information. Otherwise, if a node does not respond, **macping** will print a timeout message.

Usage:

```
/opt/scali/bin/macping [-n <ppa number>] [[<nodeid0> <nodeid1>...] |
                    [<nodeid0>:<nodeidx> [<step>]]]
```

Options>

-n <ppa num>	Access the specified **scimac** instance (physical point of attachment), instead of 0 (default).
<nodeid>	The SCI node identifier of one or more nodes to probe.
<step>	Node identifier increment.

If no node identifier **<nodeid>** is given, macping attempts to send a ping request to each of the SCI connected nodes currently known by the scimac driver (as displayed by **'/opt/scali/bin/macstat -a'**).

7.4.3 macctl - set the debug level of the scimac driver

The program **macctl** will set or get the debug level of the scimac driver. By default, all printing of debug information is disabled. When **scimac** debug is enabled, **scimac** driver information is printed to the system log file (e.g. **/var/log/messages**).

Note that if debug is enabled in the driver, it will automatically slow down the transfer of data packets, and may lead to network congestion or loss of packets.

Usage:

```
/opt/scali/bin/macctl -d [<debug level>]
```

Options:

-d <debug level> The debug level to be enabled (0 to disable debug code).

If no **<debug level>** option is specified, **macctl** prints a list of available debug level values and their meaning.

Debug level values (in any hexadecimal (0x) combination, or 0 to disable):

```
SCIMAC_WARN      0x1   /* Error conditions (WARNING style) */
SCIMAC_INFO      0x2   /* Useful information about events */
SCIMAC_EP        0x4   /* Device driver entry and exit points */
SCIMAC_PT        0x8   /* Management service requests/responses */
SCIMAC_QFULL    0x10   /* Full rcv/snd queue logging */
SCIMAC_INTR     0x20   /* Interrupt information */
SCIMAC_IPATH    0x40   /* Setup of paths and initial communication */
SCIMAC_RPATH    0x80   /* Tracking of path referencing */
SCIMAC_MSG     0x100   /* Handling of scimac messages */
```

```
SCIMAC_PMS          0x200    /* Promiscuous mode logging */
SCIMAC_PPA          0x400    /* Allocation and (de)reference of ppa */
SCIMAC_FLOW         0x800    /* Flow control logging */
SCIMAC_XFER        0x1000    /* Packet transfer logging */
SCIMAC_EBUF        0x2000    /* Ebuf usage */
SCIMAC_THREAD      0x4000    /* Information on the thread operations */
SCIMAC_ALLOC       0x8000    /* Kernel memory alloc & free operations */
SCIMAC_MEM        0x10000    /* Memory copy operations */
SCIMAC_CH         0x20000    /* Channel setup */
SCIMAC_TIME       0x40000    /* Timing of different parts of the code */
SCIMAC_INEMPTY    0x80000    /* Tracking empty receive buffer */
SCIMAC_RCVHDRS   0x100000    /* Rcv ring buffer headers (if no new pkt
                             found) */
SCIMAC_XMTHDRS   0x200000    /* Local copy of xmt headers (if remote rcv full)
                             */
SCIMAC_SCAMEM    0x400000    /* ScaMem operations */
SCIMAC_PING      0x800000    /* Probing reachability of a remote scimac
                             driver */
SCIMAC_URG      0x1000000    /* Broadcasting URG request from ScaIP */
```

7.5 ScaMAC package installation

The **scimac** driver and the ScaMAC utilities cannot be used for data transfer over SCI until the ScaMAC package (distributed as a Linux RPM file) has been installed on each processing node.

To install the ScaMAC package you *should* use the Scali Software Platform (SSP) installation program. The SSP installation program will install the specified software packages on each of the processing nodes. The ScaMAC package is automatically installed when you decide to install the ScaIP package.

The **scimac** driver depends on the **ssci** driver (Scali PCI SCI driver) included in the ScaSCI package. The ScaSCI package is automatically installed during an SSP installation. The chapter *Software Installation* in the Scali System Guide offers detailed information on how to use the SSP installation program.

Another way to install (or update) the ScaMAC package is interactively by hand using the Linux RPM package manager program **rpm**. Note that the operation should be performed on each of the processing nodes.

For a description of how to update the ScaMAC package using the **rpm** program, please consult the text file **`/opt/scali/doc/ScaMAC/INSTALL`.**

Chapter 8 Support

8.1 Feedback

Scali appreciates any suggestions to improve both this Scali Library User's Guide and the software described herein. Please send your comments by e-mail to **support@scali.com.**

The user of parallel tools software using ScaMPI on a Scali System, is encouraged to provide feedback to the National HPCC Software Exchange (NHSE) - Parallel Tools Library [9]. The Parallel Tools Library provides information about parallel system software and tools, and, in addition, it provides for communication between the software author and the user.

8.2 Scali mailing lists

We have developed mailing lists being available on the Internet. For instructions on how to subscribe to a mailing list (e.g., **scali-announce** or **scali-user**), please check out the *Mailing Lists* section at **http://www.scali.com/support**.

8.3 ScaMPI FAQ

The ScaMPI Frequently Asked Questions are posted on our Web site at **http://www.scali.com**. Please check out the *ScaMPI FAQ* section at **http://www.scali.com/support**. In addition, the FAQ is, when ScaMPI has been installed, available as a text file in **/opt/scali/doc/ScaMPI/FAQ.**

8.4 ScaMPI release documents

When ScaMPI has been installed, a number of small documents like FAQ, RELEASE NOTES, README, SUPPORT, LICENSE_TERMS, INSTALL are available as text files in the **/opt/scali/doc/ScaMPI** directory.

8.5 Problem reports

Problem reports should, whenever possible, include both a description of the problem, the software versions, the computer architecture, an example, and a record of the sequence of events causing the problem. Any information that you can include about what triggered the error will be helpful. The report should be sent by e-mail to **support@scali.com.**

8.6 Platforms supported

ScaMPI is available for a number of platforms. For up-to-date information, please check out the *ScaMPI* section at **http://www.scali.com/products**. For additional information, please don't hesitate to contact Scali at **sales@scali.com**.

8.7 Licensing

ScaMPI is licensed using the Scali license manager system. In order to run ScaMPI a valid demo or a permanent license must be obtained.

To obtain the appropriate license, please send an inquiry to **sales@scali.com**. Any technical issues should be addressed to **support@scali.com.**

Chapter 9 Related documentation

9.1 References

[1] **MPI: A Message-Passing Interface Standard**
The Message Passing Interface Forum, Version 1.1, June 12, 1995,
Message Passing Interface Forum, *http://www.mpi-forum.org.*

[2] **MPI: The complete Reference: Volume 1, The MPI Core**
Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, Jack Dongarra. 2e, 1998,
The MIT Press, *http://www.mitpress.com.*

[3] **MPI: The complete Reference: Volume 2, The MPI Extension**
William Grop, Steven Huss-Lederman, Ewing Lusk, Bill Nitzberg, W. Saphir, Marc Snir, 1998,
The MIT Press, *http://www.mitpress.com.*

[4] **Scali System Guide**
Scali AS, *http://www.scali.com/*

[5] **ScaMPI Data sheet**
Scali AS, *http://www.scali.com/*

[6] **Scali Free Tools**
Scali AS, *http://www.scali.com./*

[7] **Review of Performance Analysis Tools for MPI Parallel Programs**
UTK Computer Science Department, http://www.cs.utk.edu/~browne/perftools-review/.

[8] **Debugging Tools and Standards**
HPDF - High Performance Debugger Forum, *http://www.ptools.org/hpdf/.*

[9] **Parallel Systems Software and Tools**
NHSE - National HPCC Software Exchange, *http://www.nhse.org/ptlib.*

[10] **MPICH - A Portable Implementation of MPI**
The MPICH home page, *http://www.mcs.anl.gov/mpi/mpich/index.html.*

[11] **MPI Test Suites freely available**
Argone National Laboratory, *http://www-unix.mcs.anl.gov/mpi/mpi-test/tsuite.html*

List of figures

List of tables

Index