Scalable Computing Aachen

LEHRSTUHL FÜR BETRIEBSSYSTEME

RWTH Scalable Computing Aachen
LEHRSTUHL FÜR BETRIEBSSYSTEME
Univ.-Prof. Dr. habil. Thomas Bemmerl

SMI - Shared Memory Interface

Release 2.7

User & Reference Manual

October 2006

Joachim Worringen
Marcus Dormanns
Boris Bierbaum
Stefan Lankes

*

LEHRSTUHL FÜR BETRIEBSSYSTEME
Prof. Dr. habil. Thomas Bemmerl

COPYRIGHT NOTICE AND LICENSE AGREEMENT

RWTH Aachen, Lehrstuhl für Betriebssysteme
Univ.-Prof. Dr. habil. Thomas Bemmerl
Kopernikusstr. 16
D-52056 Aachen
Germany

Phone: +49 241 80 27634
Fax: +49 241 80 22339
eMail: contact@lfbs.rwth-aachen.de
WWW: http://www.lfbs.rwth-aachen.de

1 Purpose and Scope of SMI

The Shared Memory Interface (SMI in short), is a library implementation of an Application Programming Interface (API) for a parallel programming model based on shared memory regions. It is intended for the parallelization of existing programs and the development of new parallel programs on parallel architectures which support global shared memory in one or the other way. These include the very popular Symmetric MultiProcessors (SMPs) which are typically limited to small degrees of parallelism (2-8-way usually) and scalable multiprocessors with higher degrees of parallelism. Today, scalability is usually achieved by physically distributing the main memory across the individual compute nodes. Because of the resulting different memory access costs, depending on whether an access can be done locally or not, machines with such a design are said to have a NUMA architectures (Non-Uniform Memory Access). If additionally cache coherency is maintained across the entire machine, this architecture is more precisely entitled CC-NUMA (Cache Coherent - Non Uniform Memory Access). Often, a single compute node within a (CC-)NUMA architecture consists of a entire SMP itself. The main focus of SMI is to support the programmer in dealing with all the effects resulting from the NUMA characteristic and the heterogeneity stemming from the hierarchical architecture of (CC-)NUMAs with SMPs as compute nodes, as these are very popular and promising.

For several reasons, SMI does not provide an entire shared address space. The reasons for this are platform restrictions (not all implementations of (CC-)NUMA allow an entire shared address space) and considerations regarding the comfort of the parallelization of already existing programs. Instead, SMI is based on a replication approach with only partially shared data structures. The program code, executed in SPMD-style, is fully replicated together with all data which is not explicitly located in a shared memory region. Algorithms which work on shared data structures have to coordinate their activities, i.e. the work is decomposed and performed concurrently resulting in a speed-up due to parallel processing. Algorithms which work on replicated data perform work redundantly in parallel. There is no gain from this parallelism, but there is also no complexity from it. This enables a programmer to concentrate on parallelizing the time consuming part(s) of the program (most times just a small part), while at the same time being not concerned with complex coordination of the concurrent activities in the rest of the program.

This enables also a step-by-step parallelization. In each iteration of a parallelization work-loop, one can identify a single time consuming part of the code / data structures and parallelize it, or, if already parallelized, enhance the parallelization. This contrasts to e.g. message passing parallelization, which often forces the programmer to modify data structures due to parallelization, e.g. by performing local indexing of a part of an entire decomposed data structure and index translation for data exchange. But this enforces to parallelize the whole program at all before the program is able to run the first time in parallel.

Next to directly using SMI to create shared-memory SPMD-style parallel applications, it may also serve as a basis to implement other programming models (like MPI) or may be used to implement arbitrary services on a shared-memory cluster which fit into the programming model that its API supplies. In this way, SMI may be considered as a higher-level abstraction of the SISCI API simplifying the collaboration of distributed processes on a shared-memory cluster by offering easy-to-use services to utilize the physically distributed shared-memory.

2 Compiling and Running SMI Applications

Compilation and invocation of programs using SMI is absolutely straightforward. All necessary definitions of SMI functions, data-types and constants can be found in the include file `smi.h`. It has to be included in all user program modules which make use of SMI facilities.

2.1 Unix (Linux and Solaris)

After configuring, there are two scripts `smicc` and `smirun` in the `bin` directory of the SMI distribution. They help you to compile and run SMI applications.

2.1.1 Compiling an Application

The `smicc` script is a wrapper script for the standard compiler and can be used like the normal compile command. Basicall, it adds the SMI include path to the compiler command and links the object files with the required libraries to create an executable file.

Example:

```
smicc -DFOODEF -c smitest.c
smicc smitest.o foo.o -lX -lm -o smitest
```

will create the executable `smitest` from the given source and object files, passing the define `FOODEF` to the compiler and linking with the X and math libraries.

2.1.2 Running an Application

The `smirun` script is used to start an SMI application across multiple nodes. It uses the local remote shell command (usually `rsh` or `ssh`) to launch the processes on the remote nodes.

The `smirun` script uses a *machine file* to determine the hosts on which to run the SMI processes. Consider the following rules and limitations when creating or using such a file or look at the file `util/machines` for an example:

- Each line in the file contains exactly one host name. No comments, please.
- The `smirun` script selects the required number of hosts from the machine file from top to bottom. It launches one process on each of the hosts found.

If the option `-machinefile` is omitted from the `smirun` call, it uses the following strategy to find a machine file:

- look for `./machines`
- if not found, look for `~/.machines`
- if still not found, look for `$SMI_HOME/util/machines`
- start all processes on the current machine (from which `smirun` was started).

The general syntax of `smirun` is

```
smirun [<smirun options>] program_name [<program options>]
```

The <*program options*> are the parameters that are passed to each process launched. The following options are understood by `smirun` if supplied as <*smirun options*>:

`-np N`
: run with N processes. If this option is omitted,one process will be launched on every host in the machinefile used.

`-xterm`
: do open a separate xterm for each process, so that the output of the processes is cleanly separated. The environment variable `DISPLAY` must be set correctly on the host on which the smirun-command is issued. The environment variable `SMI_XTERM` or `XTERM` (with higher priority for `SMI_XTERM`) can be set to specify the command used to open the terminal window.

`-pager`

in conjunction with `-xterm`, `stderr` and `stdout` are piped to a pager. The environment variable `SMI_PAGER` or `PAGER` (higher priority for `SMI_PAGER`) can be used to specify the command through which the output is piped.

If you do not want to have to press a key after each page of output, you may set an appropiate environment variable to cause the used pager (`less` by default, use `-v` option of `smirun` to determine) to behave like the `tail` command. A usual way to choose is to set the environemt variable `LESS` to „`+F`" (i.e. `export LESS=+F`). Refer to the man page of you pager for detailed information on which startup commands are supported..

`-v`

be verbose on startup and print information on the startup process to stdout.

`-t`

just show the commands smirun would issue, but do not execute them („testing")

`-stderr FILE`

redirect the `stderr` stream into a file. The output of each process is written into a file named `FILE_x` where x is the SMI rank of the process. Existing output files are renamed to `'FILE~'`, empty output files are removed after termination of the application. If only one process created a non-empty output file, this file will be named `FILE` (and not `FILE_X`).

`-stdout FILE`

redirect the `stdout` stream into a file. The output of each process is written into a file named `FILE_x` where x is the SMI rank of the process. Existing output files are renamed to `'FILE~'`, empty output files are removed after termination of the application. If only one process created a non-empty output file, this file will be named `FILE` (and not `FILE_X`).

`-stdin FILE`

By default, only the `stdin` of the process with the SMI rank 0 is connected to the console from which `smirun` was invoked. If you need to supply input via `stdin` to all processes, you need to store this input in a file an used the `stdin` option: each process will get input from the file `FILE`.

`-machinefile FILE`

use `FILE` to describe the hosts on which to start the processes - see above

`-nodes NODE_0 ... NODE_NP-1`

if you do not want smirun to use the nodes (hosts) specified in any machines file, but want to specify the nodes to use on the command line, you can use the `-nodes` parameter. This parameter must be given *after* the `-np` parameter and must be followed by the corresponding number of node names. Of course, it makes no sense to use `-nodes` together with the `-machinefile` parameter (`-nodes` has the highest priority).

`-debug`

let the SMI library generate debug output - useful if you have problems which seem to be SCI related. The startup of the SMI library can be traced, and the error messages are more verbose and appear in the full context.
The library must have been configured with the `--enable-debug` option to create the debug output.

`-nolocal`

do always use SCI memory even if all processes are running on a single node. In this case, the SMI library would normally use SYS-V shared memory among the processes.

Example:

```
smirun -np 7 -machinefile /home/lassy/cluster/machines -xterm flood 50000
```

Starts the program `flood` on 7 nodes which are read from the file `/home/lassy/cluster/machines` and opens a separate xterm window for each process. The parameter `50000` will be passed to `flood`.

Notes:

- `smirun` is a `sh` script.

- `smirun` uses the local remote shell command to launch the processes on the remote hosts. Make sure that the related service is configured (i.e. via `~/.rhosts` for `rsh`) to allow remote execution without explicit authentication (entering the password).
- `smirun` needs to have write access to the `/tmp` directory of the host on which it is started.
- The processes of an SMI application need to synchronize in the startup phase using a TCP/IP port. The default port address that is used is 51069. If you need to change this address, set the environment variable `SMI_SYNCPORT` to an according value.

2.2 Windows NT

2.2.1 Compiling an Application

To compile a program on a Windows NT plattform using the Microsoft Visual C++ Compiler you have to create a new projectfile of the type *`Win32 Console Application`*. Afterwards you simply add all header and source files of your program to the project. Now you have to set up the following to make sure, that all includes and library links can be done successfully:

•in Project->Settings->C/C++:
Code Generation: set the *`Use run-time library`* field to multithreaded
Preprocessor: insert the include paths to your local installation of SMI (e.g. l:\users\sci\Shared_Memory_Interface\include) to the field *`additional include directories`*

•in Project->Settings->Link:
General: add the libraries *`smi.lib`* and *`libcimt.lib`* to the *`Objects/library modules`* field.
Input: add *`libcmtd.lib`* to the *`Ignore libraries`* field, insert the library paths to your local installation of SMI (e.g. l:\users\sci\Shared_Memory_Interface\lib) to the field *`Additional library path`*.

Now you can build the project.

2.2.2 Running an Application

[this section is out of date]

An executable can be run by using either the the frontend application located in the directory `tools/frontend`. The application only works if a special demon has been installed on all machines you want to use for the parallel processing. To do so, you can use the install Script in `/tools/frontend/demon` (NOTE: admin previleges required).

After you have started the frontend you can specify which program you want to start, and what machines to use. Make sure, that the path given to your executable can be interpreted from any machine. Using a mapped network connection as a path won‘t work. You also have to provide login information, otherwise the frontend won‘t be able to start processes on a remote machine.

Specify *`SCI Interface`* as active Plugin

2.3 Fortran Binding

It is possible to use SMI within a Fortran program in a mixed language mode. However, the function prototypes for the Fortran mode are somewhat different to take care of Fortran’s conventions regarding parameter passing. Therefore, all parameters are accepted as ‘call-by-reference’ parameters. In distinction to the C-version, where the return value is an integer, stating a possible error code, the Fortran functions are `SUBROUTINE`s and do therefore not return anything. Therefore, an additional last integer parameter is added to each function, referring to this error code.

Mixed language programming (Calling C from Fortran in this case) requires to consider naming conventions. The Unix-part coincidents with the Sun environment, in that it is assumed that the compiler in resolving open references in the linking step, assumes C-functions to end with a '_'. You have to include `include/smi_unix.f` where several constants are defined. For Windows NT (and the Microsoft compiler) we deliver `INTERFACES` for all `SMI` functions. These and the necessary constants are defined in `include/smi_win32.f` and have to be included.

3 SMI Terms and Conventions

One of the major objectives of SMI is to efficiently exploit Symmetric Multiprocessors (SMPs) within a cluster of workstations or PCs, comprising altogether a (CC-)NUMA multiprocessor. Within such an environment, it will be necessary to make a difference between a *process*, a *processor* and a *processing (or compute) node*. A processing node corresponds to a single, self-contained uni- or multiprocessor workstation or PC with a local main-memory module without any NUMA characteristic inside (not considering the different memory access latencies among the vertical memory hierarchy of L1-cache, L2-cache and main-memory). Therefore, a single processing node may contain several processors and runs its own (multiprocessor) operating system. A process is a single trace of activity together with an (partially exclusive) address space. During execution it can be migrated from one processor inside an SMP to another, according to the local operating system's scheduling strategy, but cannot migrate across processing node boundaries.

A consecutive piece of memory which is visible to all processes is called a *shared memory region*. Each address within such a region has a strict physical home on one of the nodes. If the home of a given address and that of a specific processor/process are identical, the address is said to be *local* to this processor/process, otherwise it is said to be *remote*.

Each shared memory region consists of one or more individual and usually consecutive pieces, located entirely on different compute nodes. Each such a piece is called a *shared memory segment*. Each segment of a region may have been created by a different process to which it is local. This means that the proccess to which a segment is local *exports* it, and all other processes (even on the same node) have to *import* the segment to be able to access it.

4 Initialization and Termination

`smi_error_t SMI_Init(int* argc, char*** argv)`

`int* argc`	*`number of command line parameters`*
`char*** argv`	*`command line parameters`*

Description

This function has to be called by each process before any other SMI function can be called as it initializes the shared memory environment. A redundant call of `SMI_Init()` returns an error. The function implicitly generates a global synchronization within all processes. The two parameters are those provided by the arguments to the `main()` function. These parameter should not be evaluated until `SMI_Init()` has been called.
Because of performance reasons, it is important to knwo that processes residing on the same processing node show successive process ranks. A numbering of the processes, as can be obtained by calling `SMI_Proc_rank()`, obeying this necessity is ensured within this function.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.
`SMI_ERR_NODEVICE`	Within the specified processors, there exists no possibility to deal with shared memory regions. It is required that either the processors have to belong to the same SMP or they have to be connected with a SCI network.
`1xxx`	An internal SMI error occurred.
`2xxx`	A system-call failed.

`smi_error_t SMI_Finalize()`

Description

This function must be called at the end of each process. It cleans-up the SMI environment. Each call to an SMI function afterwards results in an error. The function is not only required for a good programming style but really essential to free limited operating system resources which otherwise might stay allocated beyond the termination of the processes. Frequently termination of programs without calling `SMI_Finalize()` causes that eventually a program's necessary resources cannot be allocated any more and therefore `SMI_Init()` fails or another operation requiring these resources on this machine fails.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.

`void SMI_Abort(int return_code)`

`int return_code`	*`return code of the process`*

Description

Any process can call SMI_Abort() to enforce finalization of the SMI library and the immediate shut down of the whole applicationn. This should only be done in "emergency situations".

Return Value(s)

This function does never return.

5 I/O and Watchdog

`smi_error_t SMI_Redirect_IO(int errmode, char* errparam, int outmode, char* outparam, int inmode, char* inparam)`

int errmode	*mode flag for stderr*
*char *errparam*	*parameter for stderr redirection*
int outmode	*mode flag for stdout*
*char *outparam*	*parameter for stdout redirection*
int inmode	*mode flag for stdin*
*char *inparam*	*parameter for stdin redirection*

Description

Using this function, I/O streams (`stderr`, `stdout` and `stdin`) can be redirected to files. For each string, `...mode` specify what to do with it and `...param` is an additional parameter. Specifying `SMI_IO_ASIS` as a mode means that nothing is done with this stream. Specifying `SMI_IO_FILE` redirects a stream to a file which name has to be specified with the additional parameter. In the case of an output stream (`stderr` or `stdout`), the filename is automatically suffixed with a '`.x`' extension where `x` denotes the process rank to differ between the individual streams.

Return Value(s)

`SMI_SUCCESS`	No error occured.
`SMI_NOINIT`	This function has been called before `SMI_Init()`.

`smi_error_t SMI_Watchdog (int timeout)`

int timeout	*timeout (in seconds) to declare a non-responding process as crashed or one of the two values described below*

Description

SMI uses a watchdog mechanism to detect if any of the processes forming the application is blocked or has crashed. If this situation is detected, the local process aborts as well freeing all allocated resoucres. The threshold value which is used to decide if a process is still alive or not is set to a reasonable default value by the SMI library and usually needs not to be changed. However, in some cases (like debugging) it might be necessary to increase the threshold or to disable the watchdog. For this purpose, two special values are defined:

`SMI_WATCHDOG_DISABLE`	disables the watchdog, but the watchdog thread is still running
`SMI_WATCHDOG_OFF`	completely shuts down the watchdog, including the watchdog thread, which may be necessary to be able to create core files under Linux

For each of these options, make sure that all processes of the SMI application call this function at about the same time - if not, one of the watchdogs not yet disabled will terminate its process.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.
`SMI_ERR_PARAM`	The timeout value is illegal.

`smi_error_t SMI_Watchdog_callback (void (*callback_fcn)(void) )`

*void (*callback_fcn)(void)*	*A pointer to a function*

Description

If a certain user function should be executed on all processes in case that the application is terminated abnormally, this function can be registered via SMI_Watchdog_callback. This callback function is executed within a signal handler and should not call any SMI functions or a system function which may cause a signal itself.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.
`SMI_ERR_PARAM`	The function pointer is illegal (equals NULL).

`smi_error_t SMI_Debug (int switch)`

`int switch`	*`switch is interpreted as a boolean.`*

Description

The SMI library can generate debug and tracing output which may be useful to observe in case that problems occur. The generation of this output can be controlled by `SMI_Debug()`. If called with switch = 0, no debug output will be generated. Every other value turns on the debug output generation. The verbosity of the output depends on the debug mode which was used while compiling the library (see chapter 2).

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.

6 Information gathering about the runtime configuration

`smi_error_t SMI_Proc_rank(int* proc_rank)`

`smi_error_t SMI_Node_rank(int* node_rank)`

`smi_error_t SMI_Local_proc_rank(int* local_proc_rank)`

`int *proc_rank`	*`returned process rank of the calling process within all of them`*
`int *node_rank`	*`returned rank of the computing node, on that the calling process is located`*
`int *local_proc_rank`	*`returned process rank within the one machine`*

Description

For each process, `SMI_Proc_rank()` returns a unique number between zero and the total number of processes minus one. `SMI_Node_rank()` returns the computing node number of the node on which the calling process is located. This number is between 0 and the total number of computing nodes minus one. `SMI_Local_proc_rank()` returns the process rank of the calling process within all the processes that are executed on the same node,
Within the process numbering, `SMI_Init()` ensures that processes residing on the same computation node possess successive process ranks.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.
`SMI_ERR_NOINIT`	Function `SMI_Init()` was not called before.

`int SMI_Proc_to_node(int proc_rank, int* node_rank)`

`int proc_rank`	*`rank of a process`*
`int *node_rank`	*`rank of a computing node`*

Description

For a specified rank of a process, this function returns the rank of the computing node on wich it is executed.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.
`SMI_ERR_NOINIT`	Function `SMI_Init()` was not called before.
`SMI_ERR_PARAM`	A process with the specified process rank does not exist.

`smi_error_t SMI_Proc_size(int* proc_size)`

`smi_error_t SMI_Local_Proc_size(int* proc_size)`

`smi_error_t SMI_Max_local_proc_size(int* proc_size)`

`smi_error_t SMI_Node_size(int* node_size)`

`int* proc_size`	*`number of processes`*
`int* node_size`	*`total number of computation nodes`*

Description

`SMI_Proc_size`() and `SMI_Node_size()` return the total number of processes or the total number of computation nodes, respectively. `SMI_Local_proc_size()` returns the number of processes that are executed on the machine of the calling process. `SMI_Max_local_proc_size()` returns the maximum of these values among all machines.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.
`SMI_ERR_NOINIT`	Function `SMI_Init` was not called before.

smi_error_t SMI_First_proc_on_node (int node, int* proc)

int node	*rank of a compute node*
int proc	*returned rank of a process*

Description

This functions returns for the specified compute node the rank of the process that is executed on this node, that shows the smalles rank among all of those.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.
`SMI_ERR_NOINIT`	Function `SMI_Init()` was not called before.
`SMI_ERR_PARAM`	The value of `node` is illegal..

smi_error_t SMI_Get_node_name (char *nodename, int *namelen)

char nodename*	*returns the name of the local node (zero-terminated string)*
int namelen*	*indicates the length of the supplied char string (input/ output)*

Description

This function returns a name which identifies the node on which this process is running. The supplied char string `nodename` must be provided by the user, its lenght must be given `in namelen`. The function returns the name in `nodename,` possibly truncated up to `namelen` characters. The string is zero terminated, the actual length is also returned in `namelen`.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.
`SMI_ERR_NOINIT`	Function `SMI_Init()` was not called before.
`SMI_ERR_PARAM`	A null pointer or negative length was supplied.

smi_error_t SMI_Get_timer(int* secs, int* microsecs)

int secs*	*seconds*
int microsecs*	*microseconds*

Description

This function returns the current system time in seconds and microseconds.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.

smi_error_t SMI_Get_timespan(int* secs, int* microsecs)

int secs*	*seconds*
int microsecs*	*microseconds*

Description

This function returns elapsed time in seconds and microseconds sice the last call of either the function `SMI_Get_timer()` or `SMI_Get_timespan()`. If neither of these functions has been called before, the returned value states the timespan since program start.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.

`void SMI_Get_ticks(void* ticks)`

void ticks*	*CPU ticks (**NOTE**: ticks has to be a 64 bit integer)*

Description

This function returns the current value of the clock counter in the CPU. It can be used for high-resolution, low-overhead measurements. The actual duration of one tick depends on the CPU clock frequency which can be determined via `SMI_Query()`..

Return Value(s)

NONE

`double SMI_Wtime()`

Description

Like the well-known `MPI_Wtime()` of MPI, `SMI_Wtime()` returns the the current wall clock time (in seconds) related to some arbitrary point in the past.

Return Value(s)

The current wall clock time in seconds.

`smi_error_t SMI_Query (smi_query_t cmd, int arg, void *result);`

smi_query_t cmd	*command describing the query to perform*
int arg	*(optional) argument for the command*
*void *result*	*the result of the query (actual type depends on the query command)*

Description

SMI_Query() allows to gather system-specific runtime information. The following query types are currently supported:

SCI related queries.

SMI_Q_SCI_STREAMBUFSIZE
: retrieve the size of the streambuffers on the specified PCI-SCI adapter
arg: number of adapter to query
type of result: int

SMI_Q_SCI_PACKETSIZE
: retrieve the size of the biggest SCI packet type supported by adapter (e.g. 64 byte for LC-2 based and 128 byte for LC-3 based adapters from Dolphin)
arg: number of adapter to query
type of result: int

SMI_Q_SCI_NBRSTREAMBUFS
: retrieve the number of streambuffers on the specified PCI-SCI adapter
arg: number of adapter to query
type of result: int

SMI_Q_SCI_NBRADAPTERS
: retrieve the number of configured PCI-SCI adapters found in the local node
arg: none
type of result: int

SMI_Q_SCI_ADAPTERTYPE
: retrieve the type of the specified PCI-SCI adapter
The result is an integer which is an equivalent to Dolphin's PCI-SCI adapters model names (310, 320, 321, ...)
arg: number of adapter to query
result: int

SMI_Q_SCI_DEFADAPTER
: retrieve the number of the default PCI-SCI adapter
arg: none

type of result: int

SMI_Q_SCI_NEXTADAPTER — retrieve the number of the next usable PCI-SCI adapter (cyclic wrap-around) to import or export a region/segment. This can be used for load-distribution/load-balancing
arg: none
type of result: int

SMI_Q_SCI_ID — retrieve the SCI ID of the specified PCI-SCI adapter
arg: number of adapter to query
type of result: int

SMI_Q_SCI_PROC_ID — retrieve the SCI ID of the PCI-SCI adapter that the specified process uses to communicate with the local process
arg: rank of the remote process
type of result: int

SMI_Q_SCI_CONNECTION_STATE — retrieve the current state of the SCI connection.
The result is `SMI_SUCCESS` for a good connection or `SMI_ERR_PENDING` for a disturbed or broken connection.
arg: none
type of result: int

SMI_Q_SCI_API_VERSION — returns the version string of the SISCI API
arg: size of the memory block provided via `result` pointer
type of result: pointer to null-terminated string

SMI related queries.

SMI_Q_SMI_INITIALIZED — test if the SMI library has already been initialized
arg: none
type of result: int (to be interpreted as a boolean)

SMI_Q_SMI_REGION_CONNECTED — test if the specified shared memory region is already connected to the local process
arg: SMI shared region id
type of result: int (to be interpreted as a boolean)

SMI_Q_SMI_REGION_SGMT_ID — retrieve the internal ID of the local segment of the specfied shared memory region
arg: SMI shared region id
type of result: int

SMI_Q_SMI_REGION_ADPT — retrieve adapter number which is used to access the specfied shared memory region. A return value of -1 indicates that this region is not exported or imported, but is a SMP-local region.
arg: SMI shared region id
type of result: int

System related queries.

SMI_Q_SYS_NBRCPUS — retrieve the number of active CPUs in this node
arg: none
type of result: int

SMI_Q_SYS_CPUFREQ — retrieve the clock frequency (given in MHz) of the CPUs
arg: none
result: int

SMI_Q_SYS_PAGESIZE — retrieve the size of the pages of the virtual memory management unit of the system. If the individual machines use different size, the smalles common multiple is returned.
arg: none
result: int

Return Value(s)

`SMI_SUCCESS` — Function successfully processed.

`SMI_ERR_NOINIT`	Function `SMI_Ini()t` was not called before (does not apply for the `SMI_Q_SMI_INITIALIZED` query)
`SMI_ERR_PARAM`	A supplied parameter is illegal, the returned result is invalid.

`smi_error_t SMI_Page_size(int* psz)`

`int* psz`	*`delivered page size`*

Description

This function delivers the size of the pages of the virtual memory management unit of the system. If the individual machines use different size, the smalles common multiple is returned.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.
`SMI_ERR_NOINIT`	Function `SMI_Init()` was not called before.

7 Establishment of Shared Regions

```
smi_error_t SMI_Create_shreg(int shreg_type,
                             smi_region_info_t *shreg_desc,
                             int* id, char** address)
```

int shreg_type	*type of the region which indicates how the total requested shared memory region is physically distributed among the individual processing nodes*
*smi_region_info_t *shreg_desc*	*specific description of the shared memory region*
int id*	*returned SMI ID of the created shared memory region*
*char** address*	*returned start address(es) of the established shared memory region(s)*

Description

This function establishes a shared memory region and maps it to an address within all parallel processes' address space. The shared memory region is identified by the returned `id`-value. It's start address is returned in `address` and usually is the same for all processes to enable the exchange of pointers to addresses in this region among processes (see `SMI_SHM_NONFIXED` flag).

The physically location of the memory addresses that make up the region is specified by `shreg_type` in conjunction with `shreg_desc`. `shreg_type` can take any of the values given below. Depending on the type of the shared region, not all entries in the region description shreg_desc need to be set and their exact meaning may not the same for all types. However, two entries of shreg_desc have the same meaning for all types and need to be set accordingly:

`shreg_desc.size` The total size of the region given in bytes. This value is rounded up to the next multiple of the biggest page size (as returned by `SMI_Page_size()`) of any node on which a process of the application is running
For the process to which the region is local, this indicates the full size of the region. For a process to which the region is remote, it is the size of the region that is mapped into its address space (see the `offset` entry to understand why this is not necessarily the total physical size of the region).

`shreg_desc.adapter` A node may have multiple PCI-SCI adapters installed which are numbered consecutively, starting with 0 (see the `SMI_Q_SCI_NBRADAPTERS` query, chapter 6). This entry specifies which adapter is used to export and/or import the segments of this region. The possible values for this entry are described below.

SMI supports the efficient usage of multiple PCI-SCI adapters in one node by *adapter scheduling* strategies.. The way the available adapters are assigned to exported and imported memory regions can be controlled by the value that is assigned to the `adapter` entry of the `smi_region_info_t` that is passed to `SMI_Create_shreg`(). Please consider that the efficiency of these strategy depends on the PCI architecture of the node (single or multiple independent PCI buses) and possibly other processes which make use of the PCI-SCI adapters at the same time. Valid values are:

`SMI_ADPT_DEFAULT` All regions will be accessed via the default adapter.

`SMI_ADPT_CYCLIC` The available adapters will be used in a cyclic (round-robin) scheduled manner. I.e., if two adapters are available, the first region will be created using adapter 0, the second region using adapter 1, the third region uses adapter 0 again, and so on.

`SMI_ADPT_IMPEXP` A process will use different adapters for importing and for exporting memory (thus, this strategy is useful for two available adapters in a node). This will increase the effective bandwidth if concurrently, the local process writes to remote memory and a remote process writes to the local memory.

`SMI_ADPT_SMP` — Concurrent processes accessing a single adapter (like it may occur in a SMP node) can severely degrade the performance. This scheduling strategy tries to assign a different adapter to each local process of the SMI application. If more local processes are running than adapters are available, the assignment of the processes to the adapters will take place in a round-robin manner.

A number of flags exist that influences the way that a region is created. Not all flags are valid for all region types (refer to the detailed region type description below). Available flags are:

`SMI_SHM_DELAYED` — This flags only influences the proceses which are not owner of the region. It means that when `SMI_Create_shreg()` returns, the region can not yet be accessed as it is not yet imported. Instead, the process needs to connect to the region (and thus import it) using `SMI_Connect_shreg()` when it needs to access the region. This reduces the time required for the initial creation and saves resources (address space) on the process to which the region is remote.

`SMI_SHM_NONFIXED` — Normally, a shared memory region has the same starting address at each process which has access to it. This means that a pointer from one process pointing to location in a shared memory region is valid in another process, too. However, the technique to achieve this result which is named „fixed adressing" depends on certain resources being available in each process - this also means that it is not always possible to create a region with fixed addressing (depending on the state of the system and the operating system). As it is not always necessary to have region with fixed adressing, this flag allows to create regions with non-fixed adressing. The probability that a creation of such a region will fail is much lower, but in turn it is not possible to exchange pointers related to this region. Instead, the processes need to exchange offsets relative to the beginning of the region if they need to exchange references to memory locations situated inside a region with non-fixed addressing. *Exchanging pointers to addresses in region with non-fixed addressing will potentially corrupt data and crash processes!*

`SMI_SHM_REGISTER` — Normally, the memory range of a newly created shared memory region is added to the process' address space. For processes' to which a segment of a region is local, this memory range is mapped against the standard main memory; the memory of a remote segment is mapped against the PCI address space. However, this means that it is not possible by standard means to convert a memory range which already exists in the process' address space into a shared memory region and thereby make any part of the local process' address space globally available.
The `SMI_SHM_REGISTER` flag is intended to circumvent this limitation and allow the export of existing address space. The address of the memory range to be exported must be passed to the function via the `address` parameter.
NOT YET IMPLEMENTED due to missing driver functionality.

`SMI_SHM_PRIVATE` — The region will not be exported which means that no other process will be able to import and access this memory. The purpose of such a region is usually to use it as a source or destination for DMA operations.
NOT YET IMPLEMENTED due to missing driver functionality.

These flags need to be passed together with the type indicator as an logical OR expression (i.e. `SMI_SHM_UNDIVIDED | SMI_SHM_NONFIXED`)

With some exceptions noted below, `SMI_Create_shreg()` is a function that needs to be called collectively by all processes of the application and thus states a global synchronization point.

`SMI_SHM_UNDIVIDED` The entire shared memory region is physically located at a single processing node. The creation mode is collective.

- Relevant entries in `shreg_desc`:

  `shreg_desc.size` see above

  `shreg_desc.adapter` see above

  `shreg_desc.owner` The rank of the process to which this region is local (it will be local to all other processes running on the same node, too).

  `shreg_desc.offset` All processes to which the region is not local (all processes with a rank different from `owner)` may import only a fraction of the region. This fraction is defined via the `size` and `offset` entries. The values of these entries are rounded to the next multiple of the application's page size.

- Valid Flags:

  `SMI_SHM_REGISTER, SMI_SHM_DELAYED, SMI_SHM_NONFIXED`

`SMI_SHM_BLOCKED` The region is split as evenly as possible into as many segments as processes exists in the application. A segment *i* is physically located on the node of process *i*. The sequence of segments are mapped to the same consecutive addresses among all processes. Because the entire region can only be split at page boundaries, the splitting at page granularity is performed in a way that as few bytes as possible reside at the wrong processing node due to this given coarse granularity. The creation mode is collective.

- Relevant entries in `shreg_desc:`

  `shreg_desc.size` see above

  `shreg_desc.adapter` see above

- Valid flags:

  `None.`

`SMI_SHM_CYCLIC` The region is split into a specified number of segments , obeying the same objectives and restrictions as the `SMI_SHM_BLOCKED` policy regarding possible splittings. These segments are mapped round-robin to the processing nodes of the processes. NOT YET IMPLEMENTED. The creation mode is collective.

- Relevant entries in `shreg_desc`

  `shreg_desc.size` see above

  `shreg_desc.adapter` see above

  `shreg_desc.nbr_sgmts`

- Valid flags:

  `None.`

`SMI_SHM_CUSTOMIZED` This type allows a user-defined splitting of the region into a specified number of segments (which must be lower or equal than the number of processes in the application). Again, the exact size of the segments is system-dependent according the before mentioned objectives and restrictions regarding page boundaries. This means that the total size of the region as well as the size of each segment has to be a multiple of the page size (if this is not the case, these values are rounded up). The exact layout is specified via the `shreg_desc` parameter. Each process has to supply the same parameters. The creation mode is collective.

- Relevant entries in `shreg_desc`:

`shreg_desc.size`	see above
`shreg_desc.adapter`	see above
`shreg_desc.nbr_sgmts`	The number of segments into which the region is split.
`shreg_desc.sgmt_owner`	The mapping of the number of each segment to the rank of the process to which this segment will be local (pointer to an array: `int[nbr_sgmts]`)
`shreg_desc.sgmt_size`	The size of each segment in bytes (pointer to an array: `int[nbr_sgmts])`.The sum of the segment sizes must equal the total size of the region.

- Valid flags:

`None.`

`SMI_SHM_SMP` This type of region leads to the creation of one shared memory region on each node on which processes of the application are running. Each region is shared among the processes on the corresponding nodes, processes can not access the SMP region located on other nodes. All processes on a nodes need to supply identical parameters for the creation of a SMP region. However, the parameters may differ between the nodes. The region consists of a single segment located on each process' local node. The creation mode for this type is collective.

- Relevant entries in `shreg_desc`:

`shreg_desc.size`	see above

- Valid flags:

`SMI_SHM_NONFIXED`

`SMI_SHM_PT2PT` This type of region can be used if only two processes need to share a memory region. A PT2PT region consists of a single segment. The creation mode is non-collective; only the two processes which export and import the region need to participate on the creation or deletion of a region of this type. See the region types `SMI_SHM_LOCAL` and `SMI_SHM_REMOTE` for another type of PT2PT region.

- Relevant entries in `shreg_desc`:

`shreg_desc.size`	see above
`shreg_desc.adapter`	see above
`shreg_desc.owner`	Process rank of the process to which this region is local.
`shreg_desc.partner`	Process rank of the process which is *not* the owner.
`shreg_desc.offset`	The process which has the `partner` rank may import only a fraction of the region defined by the `offset` and the `size` entries.

- Valid flags:

`SMI_SHM_REGISTER, SMI_SHM_DELAYED, SMI_SHM_NONFIXED`

`SMI_SHM_LOCAL` A region of this type is only created and exported locally by this process. No remote process can access this region when it is first created. Remote processes need to created a region of type `SMI_SHM_REMOTE` type to import and then access this region. The creation mode of this region type is non-collective, only the local process is involved.

- Relevant entries in `shreg_desc`:

`shreg_desc.size`	see above
`shreg_desc.adapter`	see above
`shreg_desc.owner`	The process rank of the local process.

- Valid flags:

`SMI_SHM_REGISTER, SMI_SHM_PRIVATE, SMI_SHM_NONFIXED`

`SMI_SHM_REMOTE` After a process has created a region of type `SMI_SHM_LOCAL`, any other process can import this region by creating a region of type `SMI_SHM_REMOTE` and specifying the remote region which already exists. The creation mode of this region type is non-collective, only the local process is involved.

- Relevant entries in `shreg_desc`:

`shreg_desc.size`	see above
`shreg_desc.adapter`	see above
`shreg_desc.rmt_adapter`	The rank of the remote adapter that is used to export the segment. This entry can also be set to `SMI_ADPT_DEFAULT` if it is known that the remote process also uses the default adapter.
`shreg_desc.owner`	Process rank of the owner of the region (the one who created the `SMI_SHM_LOCAL` region)
`shreg_desc.offset`	The local process may import only a fraction of the region defined by the `offset` and the `size` entries
`shreg_desc.sgmt_id`	This is the *internal* identifier of the region as it is returned by a `SMI_Q_SMI_REGION_SGMT_ID` query via `SMI_Query()` which has to be performed by the *owner* of the region (*not* by the process which creates the `SMI_SHM_REMOTE` region). This means that this identifier has to be communicated in any way between the two processes involved - one way can be to use the communciation facilities described in chapter 10.

- Valid flags:

`SMI_SHM_DELAYED, SMI_SHM_NONFIXED (implicit)`

`SMI_SHM_RDMA` An RDMA (*remote DMA)* region is similar to a REMOTE region as it is a point-to-point region using a remote SCI segment, but it is usable with put/get operations only. An RDMA region has no address in the address space of the process which created it (`SMI_Create_shreg()` will return a `NULL` pointer); it's only description is the region id. The creation mode of this region type is non-collective, only the local process is involved.

- Relevant entries in shreg_desc:

`shreg_desc.rmt_adapter:`	The rank of the remote adapter that is used to export the segment. This entry can also be set to SMI_ADPT_DEFAULT if it is known that the remote process also uses the default adapter.
`shreg_desc.owner`	Process rank of the owner of the region (the one who created the `SMI_SHM_LOCAL` region)
`shreg_desc.sgmt_id`	This is the *internal* identifier of the region as it is returned by a `SMI_Q_SMI_REGION_SGMT_ID` query via `SMI_Query()` which has to

be performed by the *owner* of the region (*not* by the process which creates the SMI_SHM_REMOTE region). This means that this identifier has to be communicated in any way between the two processes involved - one way can be to use the communciation facilities described in chapter 10.

SMI_SHM_FRAGMENTED — This region type can be used if every process of the application shall export an region of type SMI_SHM_UNDIVIDED and shall also import all the region exported by the other processes. Instead of creating the the corresponding number of regions of type SMI_SHM_UNDIVIDED, it is faster and more convenient to create a single region of type SMI_SHM_FRAGMENTED. You could also interpret this region type as a region of type SMI_SHM_BLOCKED where the segments of the region are not consecuting in the address space. This region has not only one start address, but one start address for every segment. These addresses are returned via the address parameter which, in this case, *does not point to a single pointer, but to an array of pointers* (one for each process). The start adresses of the segments are not guaranteed to be identical for all processes (non-fixed addressing). The creation mode of this region type is collective.

- Relevant entries in shreg_desc:

  shreg_desc.size — Also for this type, size is the total size in bytes of the region; this means each process of the application creates a segment sized size/nbr_of_processes.

  shreg_desc.adapter — see above

- Valid flags:

  None.

Return Value(s)

SMI_SUCCESS	Function successfully processed.
SMI_ERR_NOINIT	SMI_Init() was not called before.
SMI_ERR_NO_SEG	Together with the number of shared segments already in use, there remain not enough shared segments (either because of operating system restrictions or by the restriction of a device driver which is necessary to set-up shared segments via e.g. a SCI network). The total number of shared segments via Dolphin SCI adapters limited (currently 256)..
SMI_ERR_NOMEM	The demanded shared memory region, or one of its component shared segments is too large.
SMI_ERR_PARAM	This error can have multiple reasons, please check the parameters according to the region type. Some hints: - SMI_SHM_CUSTOMIZED: the sum of the individual segment sizes must be equal to the total size of the demanded shared memory region - check if all parameters contain identical values on all processes if this is required - check if the offset is valid if it is different from 0 (offset + size must not be bigger than the physical region)

smi_error_t SMI_Init_reginfo(smi_region_info_t* region_desc, int size, int owner)

smi_region_info_t region_desc*	*pointer to an allocated region information structure*
int size	*size of the region in bytes*
int owner	*rank of the process to which the region will be local*

Description

This function serves to simplify the initialization of the information structure needed to create a shared region. It sets the entries `region_desc->size and region_desc->owner` to the values speficied by `size` and `owner.` All other entries of `region_desc` are set to default values (zero); they can be modified afterward if required.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.
`SMI_ERR_NOINIT`	`SMI_Init()` was not called before.
`SMI_ERR_PARAM`	A null pointer was passed in `region_desc`, or `size` specified a negative value, or `owner` contained an illegal rank.

`smi_error_t SMI_Connect_shreg(int id, char** address)`

`int id`	*`id of the shared memory region that has been created with the SMI_SHM_DELAYED flag`*
`char** address`	*`returned start address of the established shared memory region`*

Description

This function can only be used after a shared region has been created via `SMI_Create_shreg()` and the `SMI_SHM_DELAYED` flag.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.
`SMI_ERR_NOINIT`	`SMI_Init()` was not called before.
`SMI_ERR_NOSEGMENT`	Together with the number of shared segments already in use, there remain not enough shared segments (either because of operating system restrictions or by the restriction of a device driver which is necessary to set-up shared segments via e.g. a SCI network). The total number of shared segments via Dolphin SCI adapters is 256.
`SMI_ERR_NOMEM`	The demanded shared memory region, or one of it's component shared segments is too large.
`SMI_ERR_PARAM`	The supplied shared region id is invalid..

`smi_error_t SMI_Free_shreg(int id)`

`int id`	*`handle of a shared memory region`*

Description

A call to this function frees all resources associated with the shared memory region specified by `id`. Because all processes are affected by such an operation, analogously to the set-up of a shared memory region, this call states a global synchronization point for all collective region types. So all processes which have exported or imported a region have to call this function, before computation can proceed. After such a call, a call to another `SMI` function in conjunction with this region handle results in an error. All computations that access data previously located inside this shared memory region may cause a complete program crash since this memory region is no longer available in the address space of the process..
A call to `SMI_Finalize()` at the end of the program automatically frees all shared memory regions which still do exist.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.
`SMI_ERR_PARAM`	A shared memory region with this identifier does not exist or the identifiers of all calling processes do not refer to the same shared memory region.

`SMI_ERR_NOINIT`	Function `SMI_Init()` was not called before.

`smi_error_t SMI_Adr_to_region(char *adr, int *region_id)`

`char *adr`	*`an addres that belongs to a user-allocated shared memory region`*
`int *region_id`	*`return parameter: the region-id belonging to the address`*

Description

To the specified address `adr`, the id of the region to that it belongs is returned.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.
`SMI_ERR_PARAM`	The address does not belong to any user-allocated shared memory region.
`SMI_ERR_NOINIT`	Function `SMI_Init()` was not called before.

`smi_error_t SMI_Region_layout(int region_id, smi_rlayout_t **r)`

`int region_id`	*`id of a shared memory region`*
`smi_rlayout_t **r`	*`pointer to a structure in that the memory layout of a region is returned`*

Description

This function allows the user to gather information about the detailed layout of a shared memory region. A pointer to a structure is returned, that is allocated within the function, that contains all necessary information: the tiotal size adn address of the region, the number of segments that constitute to it and for each segment it's size, start address and the machine rank on that it is located. For the detailed definition of the structure see the `smi.h` include file.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.
`SMI_ERR_PARAM`	A user-allocated shared memory region with the specified id does not exist.
`SMI_ERR_NOINIT`	Function `SMI_Init()` was not called before.

8 Memory Management and Movement

8.1 Dynamic Memory Management

`smi_error_t SMI_Init_shregMMU(int region_id)`

int region_id	*identifier of a shared region*

Description

A call to this function initializes a memory management instance for later dynamic memory allocation within the specified region. This function has to be called before any call to `SMI_{I/C}malloc` or `SMI_{I/C}free` is performed regarding this region. It is recommended that only shared region with the distribution policy `SMI_SHM_UNDIVIDED` or `SMI_SHM_LOCAL` are used for this purpose because performance is not predictable otherwise. Because at the moment, the implemented memory manager is only capable of managing memory regions that are of size 2^1 for some *i*, just the greatest such fraction of the specified shared region is really used for dynamic memory allocation. Be aware, that also the memory manager's data structures itself are placed in this memory region and reduce the amount of memory availlable for allocation by the user. Also, after having called `SMI_Init_shregMMU()` for a shared memory region, memory of this region must not be accessed unless it was allocated via a call to `SMI_Imalloc()` or `SMI_Cmalloc()`.

This call is collective for collective regions (`SMI_SHM_UNDIVIDED` etc.) and non-collective for all non-collective regions (`SMI_SHM_LOCAL` etc.). This leads to some natural restrictions: for memory regions which are non-collective (region types LOCAL, REMOTE, PT2PT and PRIVATE), only the owner of such a region can use dynamic memory management for this region. This implies that only `SMI_Imalloc()` and `SMI_IFree()` can be used by this process, not `SMI_Cmalloc()` / `SMI_Cfree()`.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.
`SMI_ERR_NOINIT`	`SMI_Init()` was not called before.

`smi_error_t SMI_Imalloc(int size, int id, char** address)`
`smi_error_t SMI_Cmalloc(int size, int id, char** address)`

int size	*size (in bytes) of the requested memory area*
int id	*id of a shared region in that the requested piece memory is to be allocated*
*char** address*	*returned starting location of the requested area*

Description

A piece of memory of `size` bytes is dynamically allocated within the shared memory region specified by `id`. It's starting `address` is returned.
In the case of `SMI_Imalloc`, the call to this function is a pure local call from one process (individual). So it does not result in a global synchronization point, nor does it require any cooperation of other processes. Invocation of this function from several processes at the same time may result in performance degradation, because some degree of mutual exclusion is necessary for correctness. In contrast, `SMI_Cmalloc` is a collective call to that all processes must participate. In this case, all parameters have to be identical. This function returns to all processes with the common address to the allocated piece of memory. If the region has not been created with the `SMI_SHM_NONFIXED` flag, the memory is visible for all processes at the same virtual address, such that also pointers can be exchanged between processes. This is more comfortable in the case that all processes need to access the allocated memory. A notification of other processes by a single caller to a malloc function about the address in no longer necessary.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.

SMI_ERR_NOINIT	SMI_Init() was not called before.
SMI_ERR_PARAM	A region with the specified id does not exist or is not initialized for dynamic memory allocation.
SMI_ERR_NOMEM	The specified segment of the specified shared memory region does not contain a free area of the requested size to satisfy the request.

smi_error_t SMI_Ifree(char* address)
smi_error_t SMI_Cfree(char* address)

char address*	*starting address of the memory area to be erased*

Description

If a piece of memory had been allocated, it is set free, otherwise the call to this function has no consequence.
In the case of SMI_Ifree() just the calling process (individually) frees the memory. This might be dangerous if other processes still keep the pointer. An error might occur if one of these other processes uses the memory afterwards or frees it itself. In contrast, SMI_Cfree() is a function that requires to be collectively be called from all processes. However, the result not different, but this induces an implicit synchronization. This makes sure when the corresponding piece of memory can be used and when no longer.

Return Value(s)

SMI_SUCCESS	Function successfully processed.
SMI_ERR_NOINIT	Function SMI_Init() was not called before.
SMI_ERR_PARAM	The address is not valid.

8.2 Memory Movement

Although the standard memcpy() function can be used to transfer memory in a SMI application, SMI provides it's own memory copy function to make the best use of the underlying hardware. The functions provided by SMI achieve a higher bandwidth than memcpy() for remote reading or writing of data, and they allow asynchronous memory copying without CPU load.

smi_error_t SMI_Memcpy (void *dest, void *src, int size, int flags)
smi_error_t SMI_Imemcpy (void *dest, void *src, int size,
int flags, smi_memcpy_handle* h)

*void *dest*	*destination address*
*void *src*	*source address*
int size	*size of memory block in bytes*
int flags	*flags to indicate memory types or specify the operation*
*smi_memcpy_handle *h*	*handle for asynchronous operation*

Description

The parameters dest, src and size are used just like memcpy() does. SMI can automatically determine the memory types involved in the operation. To avoid this overhead, it is possible to indicate the type of memory via the flags parameter as follows:

SMI_MEMCPY_*SRCTYPE_DESTTYPE*SRCTYPE and DESTTYPE have to replaced by one of the following strings:

LP for local, non-shared (private) memory
LS for shared memory on the local machine
RS for shared memory on a remote machine
Example: SMI_MEMCPY_LP_RS for copying from local private memory to remote shared memory.

Some additional flags (which have to be combined via a logical OR opereration) can be used. Be careful with these flags and only use them if you know what you are doing.

`SMI_MEMCPY_NOBARRIER`	Do not execute a store barrier after the copy operation
`SMI_MEMCPY_NOVERIFY`	Do not verify the correct transmission of the data.
`SMI_MEMCPY_ENQUEUE`	Do not immeadelety start the data transfer. The transfer request is enqueued to wait for more transfer requests. The last of the desired requests should call `SMI_Imemcpy`() *without* this flag to actually start the transfer.

The `SMI_Imemcpy`() function is executed asynchronously, returning directly after invocation with a handle. This handle is to be used to wait for the completion of the copy operation using a `SMI_Memwait`() or `SMI_Memtest`() function.
Note: Real asynchronous operation (without CPU load) of `SMI_Imemcpy()` is currently only possible for an `SMI_MEMCPY_LS_RS` type of operation.

Return Value(s)

`SMI_SUCCESS`	copy operation successfully completed (for `SMI_Memcpy()`) or posted (for `SMI_Imemcpy()`)
`SMI_ERR_PARAM`	an illegal flag was supplied

`smi_error_t SMI_Put (int region_id, int offset, void *src, int size)`

`int region_id`	*`SMI region id of the destination region`*
`int offset`	*`Offset [bytes] inside the destination region`*
`void *src`	*`source address of the data`*
`int size`	*`size of the data to be transfered`*

Description

`SMI_Put()` is intented to be used with regions of type RDMA for fast remote data transfer via DMA, without the overhead of completely mapping a remote segment into the local address space (duration for the mapping is relative to the size of the remote segment). The source address must be located in a local SCI segment (or a SCI-registered user-allocated buffer). Both source address and destination offset must be aligned to an 8 byte boundary. For best performance, 64 byte alignment is recommended.

Return Value(s)

`SMI_SUCCESS`	The data has bee transfered and the source buffer can be reused.
`SMI_ERR_PARAM`	Illegal region id; invalid size or offset specified.
`SMI_ERR_BADADR`	The source address is not located in a local SCI segment. Source address or destination offset have an illegal alignment.

`smi_error_t SMI_Get (void *dest, int region_id, int offset, int size)`

`int region_id`	*`SMI region id of the source region`*
`int offset`	*`Offset [bytes] inside the source region`*
`void *src`	*`destination address of the data`*
`int size`	*`size of the data to be transfered`*

Description

`SMI_Get()` is intented to be used with regions of type RDMA for fast remote data transfer via DMA, without the overhead of completely mapping a remote segment into the local address space (duration for the mapping is relative to the size of the remote segment). The destination address must be located in a local SCI segment (or a SCI-registered user-allocated buffer). Both destination address and source offset must be aligned to an 8 byte boundary. For best performance, 64 byte alignment is recommended.
Note: remote read access via `SMI_Get()` will achieve a much lower performance than remote write access (`SMI_Put()`).

Return Value(s)

`SMI_SUCCESS`	The data has bee transfered and the destination buffer contains the data.
`SMI_ERR_PARAM`	Illegal region id; invalid size or offset specified.
`SMI_ERR_BADADR`	The destiniation address is not located in a local SCI segment. Destination address or source offset have an illegal alignment.

`smi_error_t SMI_Memwait (smi_memcpy_handle h)`

`smi_memcpy_handle h` *`a handle referencing the memcpy operation`*

Description

This function waits for the completion of the related memory copy operation. It does not use CPU cycles during this wait.

Return Value(s)

`SMI_SUCCESS`	copy operation successfully completed
`SMI_ERR_PARAM`	an illegal handle was supplied
`SMI_ERR_TRANSFER`	sequence error during transfer - transfer has to be repeated
`SMI_ERR_NOTPOSTED`	the transfer related to the handle has been enqueued, but the transmission of the request was not yet started.

`smi_error_t SMI_MemwaitAll (int count, smi_memcpy_handle *h, smi_error_t *status)`

`int count`	*`number of handles in the array`*
`smi_memcpy_handle *h`	*`array of handles referencing the memcpy operations`*
`smi_error_t *status`	*`array of memcpy operation statuses`*

Description

`SMI_MemwaitAll()` is an extension of `SMI_MemWait()` to wait for multiple outstanding copy operations. If the return code does not equal `SMI_SUCCESS`, the array of statuses contains a return code for each individual operation.

Return Value(s)

`SMI_SUCCESS`	all memory transfers have completed successfully
`SMI_ERR_OTHER`	look at the individual status variables to find the exeact reason

`smi_error_t SMI_Memtest (smi_memcpy_handle h)`

`smi_memcpy_handle h` *`a handle referencing the memcpy operation`*

Description

`SMI_Memtest()` tests for the completion of the copy operation related to the given handle. It returns immeadeletely, indicating the status of the operation with its return code.

Return Value(s)

`SMI_SUCCESS`	memory transfer has completed successfully
`SMI_ERR_PENDING`	memory transfer has not yet completed
`SMI_ERR_PARAM`	illegal handle was supplied
`SMI_ERR_TRANSFER`	sequence error occured during transfer - transfer has to be repeated
`SMI_ERR_NOTPOSTED`	the transfer related to the handle has been enqueued, but the transmission of the request was not yet started.

`smi_error_t SMI_MemtestAll (int count, smi_memcpy_handle *h,`

smi_error_t *status)

int count	*number of handles in the array*
smi_memcpy_handle h	*a handle referencing the memcpy operation*
*smi_error_t *status*	*array of memcpy operation statuses*

Description

`SMI_MemtestAll()` is an extension of `SMI_Memtest()` to check for the status of multiple outstanding copy operations. If the return code does not equal `SMI_SUCCESS`, the array of statuses contains a return code for each individual operation.

Return Value(s)

`SMI_SUCCESS`	memory transfer has completed successfully
`SMI_ERR_OTHER`	check the `status` array for individual error codes
`SMI_ERR_PARAM`	illegal handle was supplied

smi_error_t SMI_Check_transfer(int flags)

int flags	*flags to select the consistency operations which are performed for the check*

Description

Note: This function is not designed for general use. Instead, the verification of transfers is done by the SMI memory copy functions.
If a manual verification of a memory transfer is required, `SMI_Check_transfer`() indicates if any errors have occured since the last call to `SMI_Check_transfer`(). The following flags can be used:

SMI_CHECK_FULL	Flush all buffers and wait for completion of outstanding transfers. This is the default.
SMI_CHECK_NOFLUSH	Checks without flushing any buffers.
SMI_CHECK_NOBARRIER	Does not wait for outstanding transfers (no store barrier).
SMI_CHECK_FAST	Combination of SMI_CHECK_NOFLUSH and SMI_CHECK_NOBARRIER.

9 Synchronization

`smi_error_t SMI_Barrier()`

Description

This most restrictive synchronization service performs a barrier synchronization between all parallel processes.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.
`SMI_ERR_NOINIT`	Function `SMI_Init()` was not called before.

9.1 Mutex synchronization

`smi_error_t SMI_Mutex_init(int* id)`
`smi_error_t SMI_Mutex_init_with_locality(int* id, int prank)`
`smi_error_t SMI_Mutex_destroy(int id)`

`int(*) id`	*`returned identifier of the installed mutex`*
`int prank`	*`process rank of one of the processes`*

Description

These functions install a mutex or destroys it, respectively. To install a mutex, one of the two functions `SMI_Mutex_init()` and `SMI_Mutex_init_with_locality()` can be used. The first one distributes the necessary data structures regularly among all compute nodes, if possible and meaningfull. This results in a (more or less) evenly distributed lock/unlock performance, regardless which process uses the mutex. In the case that it is obvious that a mutex is used mostly by a single process, optimizations are possible in that all the mutex's data structures are located on these process' compute node. This allows the distinguished process to have a very fast acces to the mutex, comming along with the effect that all other have a somewhat more expensive one to it. If such a behaviour is desired, `SMI_Mutex_init_with_locality()` should be used, specifying the process rank of that process at whose compute node the data structures are to be allocated in the parameter `prank`.
To destroy a mutex, it is sufficient just to pass it's `id` to `SMI_Mutex_destroy()`.
All these functions have to be called collectively by all processes, whether all of them really uses this mutex or not.

Return Value(s)

`SMI_SUCCESS`	Function successfully executed.
`SMI_ERR_PARAM`	A mutex with the specified `id` does not exist.

`smi_error_t SMI_Mutex_lock(int id)`
`smi_error_t SMI_Mutex_unlock(int id)`

`int id`	*`identifier of the corresponding mutex`*

Description

A call a a process to one of these functions locks/unlocks the specified mutex. Note: (1) the underlying algorithm implements a spin-lock, (2) the lock is free upon initialization, (3) the lock has to bee unlokked by the same process that locked it.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.
`SMI_ERR_PARAM`	A mutex with the specified `id` does not exist.

smi_error_t SMI_Mutex_trylock(int id, int* result)

int id	*identifier of a mutex*
int result*	*result of the operation*

Description

This function checks the state of the specified mutex. If the mutex is currently free, it is locked and `result` is set to `1`. Otherwise, `result` is set to `0` and the function returns without blocking the calling process.

Return Value(s)

SMI_SUCCESS	Function returned successfully.
SMI_ERR_PARAM	A mutex with the specified `id` does not exist.

9.2 Progress counter

smi_error_t SMI_Init_PC(int* pc_id)

int pc_id*	*returned identifier of a progress counter*

Description

With this function, a new progress counter can be allocated. It's identifier, that can be used to refer to it afterwards, is returned in `pc_id`.
A progress counter is a way to inform other processes of computational progress. It is an integer quantity for each process, that is initialized to 0 within this function call. This variable is guaranteed to be non-decreasing. The semantic is that a process whose progress counter states a specific values has made computational progress at least to the state that this value refers to, but maybe also even further.
This function has to be called collectively from all processes and states a global synchronization point.

Return Value(s)

SMI_SUCCESS	Function returned successfully.
SMI_ERR_NOMEM	There is not enough memory to allocate the counter variables internally.

smi_error_t SMI_Reset_PC(int pc_id)

int pc_id	*identifier of a progress counter*

Description

Resets all processes' values of the specified progress counter to zero. This is a collective function call that states a global synchronization point.

Return Value(s)

SMI_SUCCESS	Function returned successfully.
SMI_ERR_PARAM	A progress counter with the specified `pc_id` does not exist.

smi_error_t SMI_Increment_PC(int pc_id, int val)

int pc_id	*identifier of a progress counter*
int val	*incrementation size*

Description

This function increments the own value of a specified progress counter by `val`.

Return Value(s)

SMI_SUCCESS	Function returned successfully.
SMI_ERR_PARAM	A progress counter with the specified `pc_id` does not exist.

smi_error_t SMI_Get_PC(int pc_id, int proc_rank, int* pc_val)

int pc_id	*identifier of a progress counter*
int proc_rank	*rank of a process*
int pc_val*	*returned value of a progress counter*

Description

This function returns the value of a specified process within a specified progress counter.

Return Value(s)

SMI_SUCCESS	Function returned successfully.
SMI_ERR_PARAM	A progress counter with the specified `pc_id` or a process with the rank `proc_rank` does not exist.

smi_error_t SMI_Wait_individual_PC(int pc_id, int proc_rank, int pc_val)

smi_error_t SMI_Wait_collective_PC(int pc_id, int pc_val)

int pc_id	*identifier of a progress counter*
int proc_rank	*rank of a process*
int pc_val	*value of a progress counter*

Description

This function waits until some counter values of processes within the specified progress counter have reached a certain value. With the function `SMI_Wait_individual_PC()`, it is waited just for a single process, with the rank `proc_rank`. With a call to `SMI_Wait_collective_PC()`, it is waited untill all processes have reached a specific progress. The value until that the function shall wait is specified with `pc_val`. Besides specifying a concrete value, one can specify the constant `SMI_OWNPC`. In this case, the own progress value is taken.

Return Value(s)

SMI_SUCCESS	Function returned successfully.
SMI_ERR_PARAM	A progress counter with the specified `pc_id` does not exist.

9.3 Signalization

smi_error_t SMI_Signal_wait(int proc_rank)

int proc_rank	*rank of the process from which the signal is expected*

Description

This function blocks until the process `proc_rank` signalizes the local process by sending a signal via `SMI_Signal_send()`. The waiting for a signal does not cost CPU cycles.

The following value can also be used for the process rank:

SMI_SIGNAL_ANY	accept a signal from any process

Return Value(s)

SMI_SUCCESS	Function returned successfully.
SMI_ERR_PARAM	A process with the specified `rank` does not exist.

smi_error_t SMI_Signal_send(int proc_rank)

int proc_rank	*rank of the process which is to be signalized*

Description

This function signalizes the process `proc_rank` by sending a signal. The signaled process will return

from a call to `SMI_Signal_wait()` or will execute the callback function (if set) upon reception of the signal. If the signaled process is not currently waiting for a signal, the signal will be lost.
The following value can also be used for the process rank:

`SMI_SIGNAL_BCAST`	send a signal to all other processes

Return Value(s)

`SMI_SUCCESS`	Function returned successfully.
`SMI_ERR_PARAM`	A process with the specified `rank` does not exist.

`smi_error_t SMI_Signal_setCallBack (int proc_rank, void (*callback_fcn)(void *), void *callback_arg, smi_signal_handle* h)`

int proc_rank	*rank of the process from which the signal is expected*
*void (*callback_fcn)(void *)*	*a pointer to the function which is to be called when the signal arrives*
*void *callback_arg*	*optional argument pointer for callback_fcn*
*smi_signal_handle *h*	*a handle by which the callback can be joined*

Description

This function sets a callback function for the arrival of a signal: `callback_fcn`() will be processed if the indicated process `proc_rank` sends a signal to the local process.. The call to `SMI_Signal_setCallBack()` returns immedeately. The calling process can wait for the callback function to have completed by using `SMI_Signal_joinCallBack()`.

The following value can also be used for the process rank:

`SMI_SIGNAL_ANY`	accept a signal from any process

Return Value(s)

`SMI_SUCCESS`	Function returned successfully.
`SMI_ERR_PARAM`	A process with the specified `rank` does not exist.

`smi_error_t SMI_Signal_joinCallBack ( smi_signal_handle* h )`

*smi_signal_handle *h*	*a handle indicating the callback to be joined*

Description

This function wait for the callback function (which was set up via a call to `SMI_Signal_setCallBack()`) to have completed. This wait does not cost any CPU cycles.

Return Value(s)

`SMI_SUCCESS`	Function returned successfully. Invalid signal handle
`SMI_ERR_PARAM`	A process with the specified `rank` does not exist.

10 Inter-Process Communication

SMI is not a message-passing library and has no intentions to support this programming model directly. However, it is sometimes useful to be able to send small amounts of data from one process towards another process or to exchange data with another process without creating and managing data structures in shared memory only for this purpose. Therefore, SMI offers a *very limited set of functions* to perform basic interprocess communication in a message-passing style. It is important to understand that this functionality can not be compared with specialized message-passing libraries like MPI, and although the function names resemble the MPI function names, the semantics of the SMI functions is different and much more limited.

`smi_error_t SMI_Send(void *buf, int count, int dest)`

`smi_error_t SMI_Isend(void *buf, int count, int dest)`

`void *buf`	*`pointer to send buffer`*
`int count`	*`number of bytes to send from send buffer`*
`int dest`	*`rank of destination process`*

Description

A process can send another process a message using `SMI_Send()` or `SMI_Isend()`. The maximum amount of data that can be transferred is defined as `SMI_MP_MAXDATA` (which is 64 byte less two times the size of an integer, resulting in 56 bytes on the supported platforms). When the function return with `SMI_SUCCESS`, the data has been placed in the receive buffer of process `dest` and the local buffer can be reused.

If `SMI_Send()` returns successfully, the receiving process is guaranteed to have read the message (synchronous send mode). This is not true for `SMI_Isend()` (asynchronous send mode) returning successfully: the receiving process may or may not have read the message. `SMI_Send_wait()` has to be used to ensure that the receiving process has read the message and is ready to accept a new message from the calling process. Before sending a new message, the destination process must have read any previous message.

Return Value(s)

`SMI_SUCCESS`	No error occured, local send buffer can be reused.
`SMI_NOINIT`	The function has been called before `SMI_Init()`.
`SMI_PARAM`	The amount of data to be sent is to big (`count > SMI_MP_MAXDATA`)
`SMI_PENDING`	Another asynchronous send operation towards process `dest` is still in progress.

`smi_error_t SMI_Send_wait(int dest)`

`int dest`	*`rank of process which is to read the message sent`*

Description

After an asynchronous send operation, a call to `SMI_Send_wait()` does not return until the receiving process `dest` has read the message from the receive buffer. This also means that the process is ready to receive another message from the calling process..

Return Value(s)

`SMI_SUCCESS`	No error occured.
`SMI_NOINIT`	The function has been called before `SMI_Init()`

`smi_error_t SMI_Recv(void *buf, int count, int dest)`

`void *buf`	*`pointer to receive buffer`*
`int count`	*`size of message to be received`*
`int dest`	*`rank of source process`*

Description

Upon successful completion, the receive buffer `buf` contains a message of length `count` sent by process `dest`. `SMI_Recv()` does not return until a message has been sent by proces `dest`.

Return Value(s)

`SMI_SUCCESS`	No error occured
`SMI_NOINIT`	The function has been called before `SMI_Init()`.
`SMI_PARAM`	The amount of data to be received is to big (`count > SMI_MP_MAXDATA`)

`smi_error_t SMI_Sendrecv(void *send_buf, void *recv_buf, int count, int dest)`

`void *send_buf`	*`pointer to send buffer`*
`void *recv_buf`	*`pointer to receive buffer`*
`int count`	*`number of bytes send and receive`*
`int dest`	*`rank of destination process`*

Description

Calling `SMI_Sendrecv()` is identical to calling a sequence of `MPI_Isend(); MPI_Recv; MPI_Send_wait()`.

Return Value(s)

`SMI_SUCCESS`	No error occured
`SMI_NOINIT`	The function has been called before `SMI_Init()`.
`SMI_PARAM`	The amount of data to be received is to big (`count > SMI_MP_MAXDATA`)

11 Switching between different consistency states

smi_error_t SMI_Switch_to_replication(int id, int mode, int param1, int param2, int param3)

int id	*identificator of a shared memory region*
int mode	*specifies the replication mode*
int param1, param2	*specifies the mode in more detail*
int param3	

Description

This function turns an entire shared region into a private memory region for each process. The advantage is that each process has fast, local access to this replicated region. This piece of memory is located at the same address in each process's virtual address space as the shared region was before. Due to performance considerations, it might not be necessary to copy all of the data. With the parameter `mode`, one can specify what is really necessary to be copied.

`SMI_REP_EVERYTHING` Simply copies the entire region.

`param1`, `param2` and `param3` have no meaning in this context.

`SMI_REP_NOTHING` Simply copies nothing. The shared memory region is replicated afterwards, but contains no data from the formerly shared region.

`param1`, `param2` and `param3` have no meaning in this context.

`SMI_REP_LOCAL_AND_BEYOND` It is assumed that the region contains a flat array of elements. Furthermore, it is assumed that asplitted loop iterates across it and that each process just need the elements of the array corresponding to it's own share of the index range (according to the loop splitting) and the a number of preceding and succeeding elements. Just these are actually copied to the replicated array from the shared one.

Meaning of parameters:

`param1`	number of elements in the flat array
`param2`	identifier of the split loop
`param3`	number of preceding and suceeding elements

All these modes can be or'ed with the flag `SMI_REP_ONE_PER_NODE`, indicating the all the data, at least during this replicated phases, it either just read or written but with no intersection between different processors. This allows to install a single shared segment within each SMP-node for the replicated copy of all processes on this node. This saves memory and the installation is more efficient.

This function has to be called collectively from all processes. If the specified shared region is already in state 'replicated' a call to this function has no effect.

Return Value(s)

`SMI_SUCCESS`	Function terminated successfully.
`SMI_ERR_PARAM`	A shared region with the specified identifier does not exist.

smi_error_t SMI_Switch_to_sharing(int id, int comb_mode, int comb_param1, int comb_param2)

int id	*identificator of a shared memory region*
int comb_mode	*states how to combine all the replicated data to one shared region*
int comb_param1	*specifies details of 'comb_mode'*
int comb_param2	*specifies details of 'comb_mode'*

Description

This function is the counterpart to the function `SMI_Switch_to_replication()`. It again turns the specified region into a globally shared region within all processes. Because the contents of the private memory regions might differ, it has to be stated how this is to be combined to one single globally shared region again. The parameter `comb_mode` states the methodology of this combination, the parameters `comb_param1` and `comb_param2` states further details, if required. The possibilities are:

`SMI_SHR_NOTHING` This option can be used if the replication had been performed just for performance reasons but all accesses had been read-accesses. In this case, the data in the shared region is still valid and has not to be updated by data from any replication. This is the fastest option. comb_param1 and comb_param2 have no meaning in this context.

Meaning of parameters:

`comb_param1`	none
`comb_param2`	none

`SMI_SHR_SINGLE_SOURCE` This means that the contents of the memory region of one single process provides the data for the afterwards globally shared region.

Meaning of parameters:

`comb_param1`	Rank of the process which is the data source.
`comb_param2`	none

`SMI_SHR_LOOP_SPLITTING` The specified region is considered as a flat array of elements. The array elements located in the replicated regions are joined into one shared region by iterating through a simple loop which is splitted using the SMI loop-splitting algorithms. Then, the combined shared region consists after this function of the concatenation of each process's local part according to this splitting.

Meaning of parameters:

`comb_param1` Size of the array elements

`comb_param2` Pointer to a loop-spliting to be performed (iterates once across the whole index-range of this array)

BECAUSE THIS MODE REFERS TO THE OBSOLETE LOOP-SPLITTING FUNCTIONS IT SHOULD NOT ANY LONGER BE USED

`SMI_SHR_EVERY_LOCAL` Each process contributes with it's physically local shared of the total region.

Meaning of parameters:

`comb_param1`	none
`comb_param2`	none

Furthermore, some reduction operations (i.e. commutative associative operations) are implemented with some possibilities to optimize their execution. These operate on `param1` elements of arrays that are located right at the beginning of the regions. The currently implemented operations, that can be specified in `comb_mode` are: `SMI_SHR_ADD`. This has to be or'ed (i.e. combined with '|' or '`.OR.`') with a data type that specifies the type of element. These can be `SMI_DTYPE_FIXPOINT` (`int`) or `SMI_DTYPE_FLOATINGPOINT (float)`, each in single precision (than nothing else has to be specified) or double precision (in which case this has further to be or'ed with `SMI_DTYPE_HIGHPRECISION`, to be used for `long int` or `double`).

For optimization purposes, in the case that each local replication of the vector has only a few elements that are different from the neutral element, the flag `SMI_SHR_SPARSE` can be specified.

The globally shared region to that this functions switches to, shows the same starting address as the private memory regions before. This function has to be collectively called from all processes. If the specified region is already shared, this function has no effect.

Return Value(s)

`SMI_SUCCESS`	Function terminated without any error.
`SMI_ERR_PARAM`	This error can have several reasons: shared region with the specified identifier does not exist or the specified combination mode is unknown or the parameter in conjunction with the specified combination mode is not valid.

```
smi_error_t SMI_Ensure_consistency(int id, int comb_mode,
                                   int comb_param1, int comb_param2)
```

int id	*identificator of a shared memory region*
int comb_mode	*states how to combine all the replicated data to one shared region*
int comb_param1	*specifies details of 'comb_mode'*
int comb_param2	*specifies details of 'comb_mode'*

Description

Same as `SMI_Switch_to_sharing()` but the region remains replicated.

Return Value(s)

`SMI_SUCCESS`	Function terminated without any error.
`SMI_ERR_PARAM`	This error can have several reasons: shared region with the specified identifier does not exist or the specified combination mode is unknown or the parameter in conjunction with the specified combination mode is not valid.

12 Work Partitioning and Load Balancing

--- ALL FUNCTIONS IN THIS SECTION SHOULD NOT BE USED ANY LONGER ---
--- INSTEAD, USE THE MORE ADVANCED FACILITIES OF THE NEXT SECTION ---

`smi_error_t SMI_Loop_split_init(int* loop_id)`

int loop_id*	*returned identifier of the loop and its maintained splitting*

Description

This is the initialization function for the determination of a loop splitting for parallelization,
This function does not state a global synchronization point. Nevertheless, it is required that all processes call this function.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.
`SMI_ERR_NOINIT`	`SMI_Init()` was not called before.

`smi_error_t SMI_Loop_index_range(int loop_id, int *lower, int* upper, int mode)`

int id	*identifier of a loop*
int entire_lower	*lower bound of a loop-index (sub-) range*
int entire_upper	*upper bound of a loop-index (sub-) range*
int mode	*specifies what to do*

Description

This function allows to request or set the total index range of a loop or the local share of each process respectively. The detailed functionality is specified by `mode`:

`SMI_LOOP_SET_GLOBAL`	In this mode, the parameters `lower` and `upper` determine the total index range of the entire loop
`SMI_LOOP_GET_GLOBAL`	The total index range is returned in `lower` and `upper`.
`SMI_LOOP_GET_LOCAL`	A process requests it's local share of a parallelized loop.
`SMI_LOOP_SET_LOCAL`	A process notifies the system about it's local share of a parallelized loop.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.
`SMI_ERR_PARAM`	Either a shared memory region with the identifier `id` does not exist or the loop index range is empty.
`SMI_ERR_NOINIT`	`SMI_Init` was not called before.

`smi_error_t SMI_Determine_loop_splitting(int loop_id, int mode, int param1, int param2)`

int loop_id	*identifier of a loop-splitting*
int mode	*strategy to partition the loop index range*
int param1, param2	*specify the strategy in more detail*

Description

Given that the internal data structures for a loop-splitting, specified with `loop_id`, are already initialized with the total loop bounds by a call to `SMI_Loop_index_range`(), this function computes a certain splitting into non-overlapping, consecutive ranges that altogether span the total index range. Two strategies are implemented so far:

`SMI_SPLIT_REGULAR`	The index range is split into equal-sized chunks. The variables `param1` and `param2` have no meaning.
`SMI_SPLIT_OWNER`	It is assumed, that the loop under consideration iterates sequentially across an array that is located in a shared memory region. Each element of the array is assumed to be of equal size `param1`. The identifier of the regarding shared memory region is specified in `param2`. It is assumed, that the lower bound of the total index-range of the entire loop corresponds to the element that is located right at the start address of the shared region. Then, the loop index-range is split in a way, that the local index-range fraction of each process corresponds to those array-elements that are physically located in it's machine's memory, corresponding to the physical layout of the shared memory region. If a processing node corresponds to a multiprocessor, the physical local part is split evenly among all processors of it.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.
`SMI_ERR_PARAM`	A loop-splitting with the identifier `loop_id` does not exist.
`SMI_ERR_NOINIT`	`SMI_Init()` was not called before.

`smi_error_t SMI_Loop_time_start(int loop_id)`
`smi_error_t SMI_Loop_time_stop(int loop_id)`

`int loop_id`	*`identifier of the loop under consideration`*

Description

These function measure the individual computational loads of the local part of a splitted loop, `loop_id`, in combination with the processing power of the machine. The results for each process are logged internally.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.
`SMI_ERR_PARAM`	A loop-splitting with the identifier `loop_id` does not exist.
`SMI_ERR_NOINIT`	`SMI_Init` was not called before.

`smi_error_t SMI_Loop_balance_index_range(int loop_id)`

`int loop_id`	*`identifier of a loop-splitting`*

Description

A specified loop index splitting, `loop_id`, is gracefully adapted according to the measured loads of individual processes of a former execution, in direction of a better load balancing. Iterative application of this function adapts a splitting that was performed a-priori without any data-dependent knowledge or knowledge of the power differences of the individual machine to a real balanced one, or tracks a varying load situation.

Return Value(s)

`SMI_SUCCESS`	Function successfully processed.
`SMI_ERR_PARAM`	A loop-splitting with the identifier `loop_id` does not exist.
`SMI_ERR_NOINIT`	`SMI_Init()` was not called before.

13 Loop-Scheduling

`smi_error_t SMI_Loop_init(int* id, int globalLow, int globalHigh, int mode)`

int id*	*returned id of the loop*
int globalLow	*first iteration of the loop to be scheduled*
int globalHigh	*last iteration of the loop to be scheduled*
int mode	*mode how the loop is scheduled and the iterations are initially distributed*

Description

This function initializes a loop for loop-scheduling. The loop is identified by the returned `id`-value. The first iteration of the loop is passed in `globalLow` and the last (the last to be executed) in `globalHigh`. `globalHigh` must be higher or equal `globalLow` but both can be negative.

The `mode` specifies the loop-scheduling policy:

`SMI_PART_BLOCKED`	The iterations are initially distributed as blocks among the processes. Each process executes iterations from its block until all are processed. Then this process searches for unprocessed iterations at the other processes and executes them to balance the load.
`SMI_PART_CYCLIC`	The iterations are initially distributed cyclic among the processes. Each processor receives the same amount of iterations. The process executes some iterations from its own until all are processed. Then this process searches for unprocessed iterations at the other processes and executes them to balance the load.
`SMI_PART_ADAPTED_BLOCKED`	This mode works like the `SMI_PART_BLOCKED` mode. However, the block-size is adapted form run to run if the loop is executed more than once. The block-size depends on the iterations executed by each process in the previous run of the loop.
`SMI_PART_TIMED_BLOCKED`	The iterations are distributed as blocks among the processes. Although there is no load balancing during the execution of the loop, after the execution the block-size is adapted for following runs depending on the time needed by each processor.

The different modes should be used depending on the locality of the data which is access by the iterations and the distribution of the load.

This function must be called by all processes collectively and states a global synchronization point.

Return Value(s)

`SMI_SUCCESS`	Initialization was successful
`SMI_ERR_NOINIT`	`SMI_Init()` was not called before
`SMI_ERR_NOMEM`	Not enough memory

`smi_error_t SMI_Loop_free(int id)`

int id	*id of a loop initialized with SMI_Loop_init()*

Description

A call to this function frees all resources associated with a loop. The loop is identified by its `id`. Since a call to this function affects all processes it must be called collectively.

Return Value(s)

`SMI_SUCCESS`	Function returned successfully
`SMI_ERR_NOINIT`	`SMI_Init()` was not called before
`SMI_ERR_PARAM`	A loop with the specified `id` does not exist

`smi_error_t SMI_Get_iterations(int id, int* status, int* low, int* high)`

`int id`	*`id of a loop initialized with SMI_Loop_init()`*
`int* status`	*`returned status of the scheduling`*
`int* low`	*`returned first iteration to be executed`*
`int* high`	*`returned last iteration to be executed`*

Description

This function should be placed just in front of the loop to be scheduled. The bounds of the original loop have to be changed to `low` and `high`. The loop is identified by its `id` when the function is called. After all the iterations passed by `SMI_Get_iterations()` in `low` and `high` are executed the function must be called again. Then the new iterations are executed and this procedure will be repeated until `status` is `SMI_LOOP_READY` (or `low` > `high`).

An example:

```
do {
   SMI_Get_iterations(id, &status, &low, &high);
   for (i = low; i <= high; i++) { // parallel loop
      ...
   }
} while (status != SMI_LOOP_READY);
```

If the parallel loop is executed more than once, `SMI_Get_iterations()` is a global synchronization point every first time it is called per run. So you can nest the example above in another serial loop without modification:

```
for (k = 0; k < K; k++) {
   do {
      SMI_Get_iterations(id, &status, &low, &high);
      for (i = low; i <= high; i++) { // parallel loop
         ...
      }
   } while (status != SMI_LOOP_READY);
}
```

For the mode `SMI_PART_TIMED_BLOCKED` nothing should be done between two successive executions of the entire loop since the time measurement is started and stopped in `SMI_Get_iterations`().

If the `SMI_PART_CYCLIC` mode is used the loop index (`i`) must be increased by the number of processes instead by one (`i++ --> i += proc_size`).

The two values `SMI_LOOP_LOCAL` and `SMI_LOOP_REMOTE` of `status` show if the iterations were initially assign to the process (local) or taken from an other process (remote). This parameter can be used to profile and tune the code since `REMOTE` is always more expensive.

Return Value(s)

`SMI_SUCCESS`	Function processed successfully
`SMI_ERR_NOINIT`	`SMI_Init()` was not called before
`SMI_ERR_PARAM`	A loop with the specified `id` does not exist
`SMI_ERR_OTHER`	The `mode` value was not valid (internal error)

13.1 Advanced Functions

`smi_error_t SMI_Evaluate_speed(double* speedArray)`

double speedArray*	*returned array of relative processor speeds*

Description

This function tests how quick the processes can execute an evaluation code. The slowest process is assigned "1" and all other processes are assigned a value relative to this. If the number of processes equals the number of processors the speed is also the processor speed. The speed of all processes are returned in the `speedArray` to be used by the programmer (for example for data distribution). This function has to be called by all processes.

Return Value(s)

`SMI_SUCCESS`	Function processed successfully
`SMI_ERR_NOINIT`	`SMI_Init()` was not called before

`smi_error_t SMI_Use_evaluated_speed(int id)`

int id	*id of a loop initialized with SMI_Loop_init()*

Description

The speed of the processes evaluated by `SMI_Evaluate_speed()` is used for the initial distribution of the iterations among the processes. It only makes sense with the `SMI_PART_BLOCKED` and `SMI_PART_ADAPTED_BLOCKED` (first execution) mode. By better distributing the load among the processes this function can reduce the execution time of the scheduled loop, especially if the data which is access by the iterations is distributed like the iterations (using the `speedArray` for data distribution). This function has to be called before the first call of `SMI_Get_iterations()` by all processes.

Return Value(s)

`SMI_SUCCESS`	Function processed successfully
`SMI_ERR_NOINIT`	`SMI_Init()` was not called before
`SMI_ERR_PARAM`	A loop with the specified `id` does not exist

```
smi_error_t SMI_Set_loop_param(int id, double kNew,
                          int minChunkSizeLocal,int minChunkSizeRemote,
                          int maxChunkSizeLocal,int maxChunkSizeRemote)
```

int id	*id of a loop initialized with SMI_Loop_init()*
double kNew	*new value of the chunk size control variable*
int minChunkSizeLocal	*new minimum of iterations to be executed in the local phase*
int minChunkSizeRemote	*new minimum of iterations to be executed in the remote phase*
int maxChunkSizeLocal	*new maximum of iterations to be executed in the local phase*
int maxChunkSizeRemote	*new maximum of iterations to be executed in the remote phase*
constant:	
0	*SMI_NO_CHANGE*

Description

With this function one can set some parameters of the loop-scheduling for a loop specified by its `id`.

The parameter `kNew` sets a new value for the chunk size control variable `k`. By default, `k` is set to the number of processes. If the value is smaller the chunks become larger and if the value is larger the chunks become smaller. So a larger `k` reduces the danger of load imbalance but increases the overhead.

For a smaller `k` it is vice versa.

With `minChunkSizeLocal` and `minChunkSizeRemote` the minimum number of iterations passed by a `SMI_Get_iterations()` call is specified (in local and remote phase). The default value is one. These values should be increased only if the time each iteration consumes is very small. In the same way, the maximum number of iterations can be specified with `maxChunkSizeLocal` and `maxChunkSizeRemote`.

If `SMI_NO_CHANGE` is used for a parameter this parameter remains unchanged.

Return Value(s)

`SMI_SUCCESS`	Function processed successfully
`SMI_ERR_NOINIT`	`SMI_Init()` was not called before
`SMI_ERR_PARAM`	A loop with the specified `id` does not exist

`smi_error_t SMI_Set_loop_help_param(int id, int maxHelpDist)`

int id	*id of a loop initialized with SMI_Loop_init()*
int maxHelpDist	*specifies the maximum number of processes considered in the remote phase for load balancing*

Description

To change the maximum number of processes considered in the remote phase this function is called. If a process is idle it considers other processes for work (unprocessed iterations). With `maxHelpDist` (default 16) one can specify the maximum number of processes to take influence on the scalability of the loop-scheduling and the overhead. If `maxHelpDist` is set to `SMI_HELP_ONLY_SMP` only processes on the same SMP-node are considered. Use this if the locality of data is very important for the performance.

Return Value(s)

`SMI_SUCCESS`	Function processed successfully
`SMI_ERR_NOINIT`	`SMI_Init()` was not called before
`SMI_ERR_PARAM`	A loop with the specified `id` does not exist

14 Examples

The SMI distribution contains a number of examples in the `examples` subdirectory. They serve for validation of the library functionality, performance evaluation and as examples for creating SMI applications.

To build the examples, just invoke `make` in the corresponding directory after the SMI library has been built. The following paragraphs give some more information on the purpose and the usage of the different examples.

14.1 flood

[to be written]

14.2 poisson

[to be written]

14.3 memcpy_bench

The benchmark has to be run with an even number of processes. If more then 2 processes are used, memcpy_bench makes groups of 2 processes benchmarking transfer to each other. If you want to bring your SCI-network to the limit, try using all nodes and specify -b option.

`-t <int>`	This option is used to set the Number of threads to the specified int-Value. At least one Thread is required. Each thread requires its own SCI memory space
`-n <int>`	This options sets up the number of Retries for the first test-size. It is decreased debending on the increase-type of the size.
`-s <int>`	Specifies the size of the first (smallest) transfer.
`-i <int>`	Specifies the increment of the transfer size. By default, the size is doubled each time. If `-i` is specified, the increase is added to size each time.
`-e <int>`	This option specifies the size of the last transfer.
`-c <check_type>`	This option specifies special check-types:

`nocheck`	no checks at all (default, recommended to determine peak performance)
`smi_only`	use the SMI functions for error detection, but do not verify
`verify_unchecked`	do not use the SMI functions to detect transmission errors, but verify (by comparing source and destination) if a transfer was correct (errors might occur)
`verify_checked`	use the SMI function to detect transmission errors and verify (by comparing source and destination) if a transfer was correct (no errors should occur).
`verify_details`	like `verify_unchecked`, but show the differenced between src an dst
`fail_counters`	compare the number of transmission errors indicated by the related SMI functions with the effective number of transmission errors.

`-l`	Perform a local benchmark between private memory and local SCI memory

-b	Perform benchmarks on 2 nodes in both directions at the same time (bi-directional). If -l is specified, all nodes perform local benchmarks.
-a	Test asynchronous transfers using SMI_Imemcpy(). If not specified, the default (synchronous) SMI_Memcpy() will be used
-q	Enqueue DMA-transfers.
-w	Perform write accesses to the remote node. If -l option is specified, perform writes from private memory to local SCI-memory. This is the default mode.
-r	Transfers are performed in the opposite direction than with the option -w (read access to remote memory).
-f	Force DMA-transfers even if transfer-size is to small to be recognized as efficient by SMI_Imemcpy(). This option only has an effect in combination with option -a.
-o	Use the original (internal) order of the process ranks as they are given by the machines file or the command line. This means that SMI will not reorder the processes alphabetically by the node names as it would usually do. This option allows exact specification of the transfer directions.
-m <adpt_sched>	Use multiple PCI-SCI adapters (if available). The different modes of scheduling the available adapters to the processes on a node are chosen via adpt_sched:

default	every process uses the same default adapter (this is the default mode)
cyclic	the available adapters are assigned in a cyclic manner to the regions that a process creates
impexp	one adapter is used to import memory, the other one exports memory for all processes on a node
smp	each process on a node uses if different adapter if possible (cyclic assignment)

14.4 test

[to be written]

14.4.1 helloworld

This is the most basic test for SMI which just initializes the library, queries the number of processes and the rank of the local process, prints a helloworld message and then exits.

14.4.2 regions

[to be written]

14.4.3 reglimits

[to be written]

14.4.4 replicate

[to be written]

14.4.5 sync

[to be written]

14.4.6 signal

[to be written]

15 Trouble Shooting

[update required]

- Ths program fails in the `SMI_Init()` function (UNIX):
  Check if it is possible to esthablish the required number of UNIX shared memory segments with the required size on each machine which you use to run a process of the parallel program (check chapter 2 to see how increase the limits). Furthermore, check with the `ipcs` command how many UNIX shared memory segments are already in use. If necessary, remove them with the `ipcrm` command. You can find a shell-script in the `utils` directory that removes all of them.
- Synchronization-Primitives seem not to work (UNIX and NT):
  Check if you have defined the preprocessor macro `PCI` (or `SBUS` respectively) when compiling the library.

16 Internals

This chapter intends to give the technical interested user an insight into some aspects of the internal design, techniques and algorithms of SMI.

16.1 Debug Output

[to be written]

16.2 Startup and Initialization

[to be written]

16.3 Creating a Shared Memory Segment

[to be written]

16.4 Shutdown & Watchdog

[to be written]

17 Predefined Constants and Data Structures

17.1 Return Values of SMI functions

0	SMI_SUCCESS	The function operated successfully with no error.
1	SMI_ERR_OTHER	An error occurred but it's reasons and it's consequences are not further stated.
2	SMI_ERR_NOINIT	A SMI function was called before SMI_Init was called or after SMI_Finalize was called.
3	SMI_ERR_PARAM	A SMI function was called with a wrong, not further specified, parameter.
4	SMI_ERR_BADADR	It was not possible to map a region of shared memory to the address that had been determined by SMI.
5	SMI_ERR_MAPFAILED	The mapping of a segment/region of shared memory into the virtual address space of a process failed for some reason.
6	SMI_ERR_NODEVICE	There exists no (supported) facility which could be exploited to create the shared memory regions.
7	SMI_ERR_NOSEGMENT	It was not possible to allocate another segment of shared memory.
8	SMI_ERR_NOMEM	Not enough memory.
9	SMI_ERR_NOTIMPL	Indicates that the desired functionality is not yet implemented.
10	SMI_ERR_TRANSFER	The memory transfer was not successful and has to be repeated.
11	SMI_ERR_PENDING	The memory transfer is not yet completed.
12	SMI_ERR_NOTPOSTED	The memory transfer for which the status is requested has not been posted.
1xxx		The three least significant digits correspond to the internal MPI-likecommunication facilities.
2xxx		The three least significant digits correspond to an error code, which resulted from a system call inside a SMI function.

17.2 Shared Memory Region's Physical Distribution Policies

0	SMI_SHM_UNDIVIDED	The shared memory region is entirely located at a single processing node.
1	SMI_SHM_BLOCKED	The shared memory region is as evenly as possible divided into as many contiguous parts as processing nodes exist. Part *i* is physically located at the processing node of process *i*.
2	SMI_SHM_CYCLIC	The total shared memory regions is cyclically physically distributed with a given granularity. NOT YET IMPLEMENTED
3	SMI_SHM_CUSTOMIZED	The precise division of a shared memory region into as many blocks as parallel processes exist is user-speci-

		fied.
4	SMI_SHM_SMP	For *n* nodes, *n* shared memory regions are created (one on each node). Each of these regions is shared only among the processes running on each node and thus has UMA access characteristics.
5	SMI_SHM_PT2PT	The region which is created is shared only between two processes which means that the creation of this region is non-collective.
6	SMI_SHM_FRAGMENTED	If each process has to export one shared memory region, and each process imports the corresponding shared memory region of all other processes, a single segment of type SMI_SHM_FRAGMENTED can be used instead of the related number of segments with type SMI_SHM_UNDIVIDED. However, the result is the same.
7	SMI_SHM_LOCAL	A local shared memory region is created and exported, but no other processes do yet import the region. Thus, this operation is non-collective.
8	SMI_SHM_REMOTE	To import a shared memory region that another process has exported as a region of type SMI_SHM_LOCAL, a region of type SMI_SHM_REMOTE must be used.

17.3 Shared Memory Region's Attributes

1024	SMI_SHM_DELAYED	The region is exported by the process to which it is local, but not yet imported by any process to which it is remote. These processes need to connect to this region via SMI_Connect_shreg() before they can access it. This attribute is only valid for regions of type SMI_SHM_UNDIVIDED, SMI_SHM_PT2PT and SMI_SHM_FRAGMENTED.
2048	SMI_SHM_NONFIXED	The addresses of the region are not guaranteed to be identical on all processes. Thus pointers can not be exchanged between processes, only offsets relative to the start address of the segment.
4096	SMI_SHM_REGISTER	The local memory that will be exported for this shared memory region is not added to the process' address space, but a given memory area supplied by the user is used. This attribute is only valid for regions of type SMI_SHM_UNDIVIDED, SMI_SHM_PT2PT and SMI_SHM_LOCAL. NOT YET IMPLEMENTED
8182	SMI_SHM_PRIVATE	This region will not be exported, but will only be used for internal purposes (like serving as a source or target destination for DMA operations).

17.4 Shared Memory Region Description

`smi_region_info_t` is used to specify the layout of a shared memory region that is to be created via `SMI_Create_shreg()`. Not all elements of this data type are required or meaningful for all region types. Refer to chapter 7 for further information.

```
typedef struct {
```

```
    int size;              overall size of the region (bytes) as it will appear on the local process
    int owner;             owner of the region (the region is located on the node local to this process)
    int offset;            offset for importing the region (measured from its start)
    int sgmt_id;           identifier of the region to connect to (delayed connection)
    int partner;           rank of the partner process which exports the regoin
    int nbr_sgmts;         number of segments that the region will consist of
    int *sgmt_owner;       mapping of segment number to process rank (array: int[nbr_sgmts] )
    int *sgmt_size;        size of each segment of the region (array: int[nbr_sgmts] )
    int adapter;           the rank of the local PCI-SCI adapter to import/export the region
    int rmt_adapter;       the rank of the remote PCI-SCI adapter which exports the region
  } smi_region_info_t;
```

17.5 Shared Memory Region Layout Information

`smi_rlayout_t` is used to retrieve information on an existing shared memory region using `SMI_xxx()`.

```
  typedef struct {
    char* adr;             region start address
    int size;              total region size (bytes, as it was created by the owner)
    int nbr_sgmnts;        number of comprising segments
    int *sgmt_size;        size of each segment (bytes, array: int[nbr_sgmts] )
    char **sgmt_adr;       start address of each segment (array: *char[nbr_sgmts] )
    int *sgmt_node;        physical location of a segment (rank of the node, array: int[nbr_sgmts] )
  } smi_rlayout_t;
```

17.6 Memory Copy Operations Attributes

1	SMI_MEMCPY_NOBARRIER	Do not perform a store barrier after the copy operation.
2	SMI_MEMCPY_NOVERIFY	Do not verify the success of the copy operation.
3	SMI_MEMCPY_FAST	Neither perform a store barrier nor verify the success of the copy operation
4	SMI_MEMCPY_ENQUEUE	Do not immeadeletly start the asynchronous copy operation, but only enqueue this request.
8	SMI_MEMCPY_FORCE_DMA	Transer the specified data asynchronously (using DMA) even if the size of the memory block is below the lower bound for asynchronous operations.
16	SMI_MEMCPY_ALIGN	Align the size of the memory block (increase it, if necessary) to be transfered to the next multiple of the stream buffer size.

17.7 Memory Transfer Checking

0	SMI_CHECK_FULL	A full memory transfer check is done which includes flushing all buffers and performing a store barrier. This should be the default mode as it ensures full memory consistency.
1	SMI_CHECK_NOFLUSH	The read and write buffers are *not* flushed.
2	SMI_CHECK_NOBARRIER	A store barrier is *not* performed.
3	SMI_CHECK_FAST	*Neither* a buffer flush *nor* a store barrier is performed.
4	SMI_CHECK_PROBE	Returns the current state of the memory transfer operation. In case that the connection to the target node of the transfer operation is not valid, it does *not* wait until it is valid again (like all other variants above

will do).

17.8 Query Operations

17.8.1 SCI Subsystem

SMI_Q_SCI_STREAMBUFSIZE	Size of the stream buffers on the PCI-SCI adapter
SMI_Q_SCI_NBRSTREAMBUFS	Number of stream buffers on the PCI-SCI adapter
SMI_Q_SCI_NBRADAPTERS	Number of PCI-SCI adapters found in the local node
SMI_Q_SCI_ADAPTERTYPE	Type of the specified PCI-SCI adapter
SMI_Q_SCI_ID	SCI ID of the specified PCI-SCI adapter
SMI_Q_SCI_PROC_ID	SCI ID of the primary PCI-SCI adapter that the specified process uses to communicate with processes on the local node
SMI_Q_SCI_CONNECTION_STATE	The current state of the SCI connection of the specified PCI-SCI adapter
SMI_Q_SCI_API_VERSION	The version string of the SISCI API

17.8.2 SMI States

SMI_Q_SMI_INITIALIZED	Indicates if the SMI library has already been initialized via SMI_Init()
SMI_Q_SMI_REGION_CONNECTED	Indicates if the local process is already connected to the specified shared memory region
SMI_Q_SMI_REGION_SGMT_ID	Returns the internal ID of the local segment of a shared memory region

17.8.3 Node Characteristics

SMI_Q_SYS_NBRCPUS	Returns the number of CPUs on this node.
SMI_Q_SYS_CPUFREQ	Returns the clock frequency of the CPUs of the local node
SMI_Q_SYS_PAGESIZE	Returns the page size of the virtual memory system of this node.

17.9 Addressing Flags for Signaling

SMI_SIGNAL_BCAST	Send a signal to all other processes.
SMI_SIGNAL_ANY	Accept a signal from any process.

17.10 Replication Modes

SMI_REP_EVERYTHING

SMI_REP_NOTHING

SMI_REP_LOCAL_AND_BEYOND

17.11 Sharing Modes

SMI_SHR_NOTHING

SMI_SHR_SINGLE_SOURCE

SMI_SHR_LOOP_SPLITTING

SMI_SHR_EVERY_LOCAL

17.12 Loop-Scheduling Iteration Partition Policies

1	SMI_PART_BLOCKED	The Iterations are initially distributed as blocks among the processes.
2	SMI_PART_CYCLIC	The Iterations are initially distributed cyclic among the processes.
3	SMI_PART_ADAPTED_BLOCKED	The Iterations are initially distributed as blocks among the processes (SMI_PART_BLOCKED). However, the block-size is adapted form run to run if the loop is executed more than once.
4	SMI_PART_TIMED_BLOCKED	The Iterations are distributed as blocks among the processes. Although there is no load balancing during the execution of the loop. However, after each execution the block-size is adapted for the next run depending on the time needed by each processor.

17.13 Loop-Scheduling Status

0	SMI_LOOP_READY	There are no unprocessed iterations left. For the process which received this status the loop is finished. (To be sure that all processes are finished the process should wait in a barrier)
1	SMI_LOOP_LOCAL	The process got iterations from those initially distributed to him.
2	SMI_LOOP_REMOTE	The process got iterations from those distributed to other processors (load balancing).