**SANDIA REPORT**

SAND2009-xxxx
Unlimited Release
Printed February 2009

# Seshat User Manual

Rolf Riesen Sandia National Laboratories

Sandia National Laboratories

| | |
|---|---|
| Telephone: | (865) 576-8401 |
| Facsimile: | (865) 576-5728 |
| E-Mail: | reports@adonis.osti.gov |
| Online ordering: | http://www.osti.gov/bridge |

| | |
|---|---|
| Telephone: | (800) 553-6847 |
| Facsimile: | (703) 605-6900 |
| E-Mail: | orders@ntis.fedworld.gov |
| Online ordering: | http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online |

SAND2009-xxxx
Unlimited Release
Printed February 2009

# Seshat User Manual

Rolf Riesen
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1319
rolf@sandia.gov

**Abstract**

This report presents

# Contents

# List of Figures

# List of Tables

# Preface

In the early 1990’s ...

## 1 Introduction

Basic overview of what Seshat is and what it does. Avoid much detail here.

Seshat is open source and can be obtained by contacting the author.

Publications: (Riesen 2006b) (Riesen 2006a) (Riesen 2006c) (Riesen, Vaughan, and Hoefler 2006) (Riesen and Hoefler 2006) (Riesen 2007a) (Riesen 2007b)

**Figure 1.** Seshat (Source: http://en.wikipedia.org/wiki/File:Seshat.svg)

# 2 Building the Seshat Library and Applications

Seshat is a library. During the link phase, it must be inserteted between the application and the MPI library. Seshat makes use of the MPI profiling interface. In order to use an application with Seshat, two steps must be taken:

1. Build the Seshat library
2. Link Seshat with an application

We will explain each in turn.

## 2.1 Building the Seshat Library

Currently, Seshat consists of 23 C files and one header file. They are listed with a brief explanation in Table 5. Some of these files make use of gcc and preprocessor extensions (e.g., for comments, declarations within code, anonymous variadic macros, long long ints). It is therefore necessary to use gcc to compile Seshat.

Compile each file with the `-c` option to create an object file and then create a library with the command:

```
$ gcc -c -Wall -I$MPIHOME/include -I. *.c
$ ar rs libscs.a *.o
$
```

The environment variable `$MPIHOME` needs to be set to the location of the MPI library and include files. Often that is `/usr/local`.

Currently, there are three network models: `scs_network_liberty.c`, `scs_network_redstorm.c`, and `scs_network_tbird.c`. The file `scs_network_liberty.c` contains a model for the decommissioned Liberty cluster which used a Myrinet network. The file `scs_network_redstorm.c` contains a model for the network of Sandia's Red Storm network when running the Catamount operating system. Similarly, the file `scs_network_tbird.c` contains a model for the Infiniband network of the Thunderbird cluster.

Only one of these three network models needs to be included in the library.

## 2.2 Linking Seshat with an Application

The `libscs.a` built in the previous section needs to be linked with the object files of an application before it is linked with the MPI library. This has to be done so Seshat can insert itself between the applications MPI calls and the MPI library.

MPI provides a standard interface, called the MPI profiling interface, for this purpose. The MPI library is built using weak symbols. When an application calls `MPI_Send()` for example, it calls that routine in the `libscs.a` Seshat library. Seshat then calls `PMPI_Send()` to pass control flow to the actual MPI library and let it transmit data over the network. This allows Seshat to execute code before and after the actual MPI send. For example, in can measure the time, send and event, and adjust the virtual time everytime the application makes an MPI send call.

Often `mpicc` is used to compile and link MPI appplications. The problem is that the `mpicc` script makes it sometimes difficult to insert the Seshat library before the MPI library. Therefore, linking should be done like this:

```
$ gcc app.c -L. -lscs -L$MPIHOME/lib -lmpi -lrt
$
```

Libraries that are listed earlier on the command line get linked before libraries later on the command line. The above example makes sure `libscs.a` gets linked before the MPI library. The `-lrt` switch brings in the realtime library that is necessary on some systems for the `clock_gettime()` function with the `CLOCK_REALTIME` option which Seshat uses to measure time.

### 2.2.1 Application Requirements

Seshat has been tested with Fortran and C applications. Only one C++ application has been tests, and it used the MPI language bindings for C, not C++: calling `MPI_Send()`, not `MPI:Send()`.

Seshat manipulates the virtual time on each node to simulate a machine with a different network. Applications that use `MPI_Wtime()` will preceive Seshat's virtual time frame and report results accordingly. For example, a benchmark using `MPI_Wtime()` will report timings that correspond to running the same benchmark on the simulated machine. If an application uses another function to measure time, it will see the wallclock time and Seshat cannot influence its timing behavior.

Some less frequently used MPI calls do not have Seshat wrappers yet (for example `MPI_Intercomm_merge()`). All MPI functions are intercepted by Seshat. The ones with missing wrappers will print an error message and abort the program. Most wrappers are simple to write and can be based on exisiting wrappers, but that work is not complete yet.

# 3 Running Seshat

This will be mostly about the configuration file.

## 3.1 Running Seshat

Start application as before with any command line options necessary for the application, but add one to the number of nodes needed by the application.

```
$ mpirun -n 5 app
$
```

## 3.2 The Seshat Configuration File

During initialization, when the application calls `MPI_Init()`, Seshat looks for a file named `scs_conf` in the current directory. If the environment variable `SCS_CONF_FILE_NAME` is set, the file name stored there is used instead. If that file exists, Seshat opens it and reads the configuration information from it. Various parameters and options can be set in the configuration file, but none are mandatory. Seshat provides defaults for all of them, and then configuration file can be empty or non-existent.

The configuration file uses a simple, human readable format and has these properties:

- Anything after a # sign to the end of the line is ignored.
- Blank lines are ignored.
- Keywords and arguments are separated by white space.
- Keywords and arguments are case sensitive.
- White space in file names is not permitted.
- Space and tabs are considered white space.
- Adjacent white space is the same as a single space.
- Only one keyword and its argiments per line.
- The order of keywords is not important.
- The last value assigned to a keyword in the file overwrites all previous ones.

Several aspects of Seshat can be controled by related keywords. Currently, there are four categories: Debugging, simulation output, network parameters, and trace data.

**Table 1.** Debug keywords in configuration file

| Keyword | Value | Description |
|---|---|---|
| `debug` | `config` | Configuration file processing information |
| | `process` | Seshat and application startup information |
| | `mpi` | Generic information about MPI calls made by the application |
| | `mpi2` | Information about all MPI calls the application makes |
| | `mpi3` | Information about non-C MPI calls; e.g., Fortran |
| | `mpiint` | Information about simulator internal use of MPI; e.g. transmitting events |
| | `general` | generic information such as which system is being simulated and calls to non-implemented MPI functions |
| | `event` | Information about event processing |
| | `emu` | Emulator related information |
| | `net` | Network simulator information |
| | `time` | Virtual time realted information |
| | `req` | Information about Seshat's handling of non-blocking MPI requests |
| | `all` | Turn all the above options on |
| `debug_file` | *`file name`* | Place the selected debug information into the specified file |

### 3.2.1 Debugging

There are several debugging options that can be set to help debugging Seshat itself. Some of the debugging information might be interesting when you are trying to understand the MPI behavior of an application. Table 1 shows the available options.

The file name specified for `debug_file` can be `stderr` or `stdout`. The default output file descriptor is `stderr`. No debug output is selected by default.

The following fragment shows an example where timing and network information is collected into a file named `debug.out`.

```
debug net
debug time # Collect timing information

# Specify the output file
debug_file debug.out
```

### 3.2.2 Simulation Output

Seshat collects a number of statistics. If an output file for a given set of statistics is specified, then that information is written during the `MPI_Finalize()` call. If no file is specified, then Seshat

**Table 2.** Output file naming in configuration file

| Keyword | Value | Description |
|---|---|---|
| `vtime_file` | *`file name`* | Virtual time statistics (mostly for debugging) |
| `mdd_file` | *`file name`* | The message density array |
| `ddd_file` | *`file name`* | The data density array |
| `col_file` | *`file name`* | Collective operations information |
| `p2p_file` | *`file name`* | Point-to-point operations information |
| `msize_file` | *`file name`* | Message size distribution |

will not write that information (default). In addition to providing a file name to store the data, the keyword `net_reporting` has to be set as well. It serves as a on/off switch for most of the data written into the files below.

File names must not contain white space, and `stderr` and `stdout` may be specified. Table 2 lists the available keywords to direct Seshat to write statistics.

The message density array shows how many messages each rank has sent to every other rank. The data density array is similar, but it records the amount of data that has been sent in $10^6$ bytes. Each file contains a two-dimensional array. The first row in that array lists how many messages (or megabytes) rank zero has sent to rank 0, 1, 2, . . . The second row shows the corresponding numbers for rank 1, and so on. That information can be uses to analyze the message passing pattern an application exhibits.

Figure 2 shows an example of how this data can be used. After running the NAS parallel benchmark SP on 64 nodes, the file containing the message density array data can be used to plot a graph that shows which ranks sent how many messages to each other.

The example shows that the number of messages (about 3,500) exchanged bewteen ranks is always the same. Only specific ranks communicate with each other. For example, rank 0 sends data to rank 1, 7, 8, 15, 56, and 57. It receives data from the same set of nodes.

Seshat records the number of times a specific MPI collective operation is called (`MPI_Reduce()`, `MPI_Allreduce()`, etc.) in the file specified after the `col_file`.

### 3.2.3 Network Parameters

Seshat currently uses a model to simulate the network. A true simulator could be used, but at the moment, only a fairly simple model has been used with Seshat. During compilation (Section 2.1) of the Seshat library `libscs.a` on of three models must be chosen. The Red Storm model mimics the behavior of the Cray XT-3 Red Storm network running Cray Unicos/lc 1.5.39. There are also the liberty model which is a Myrinet cluster, and the Tbird model which is an infiniband cluster.

Each of these models encompases a point-to-point model and a collective model. A collective

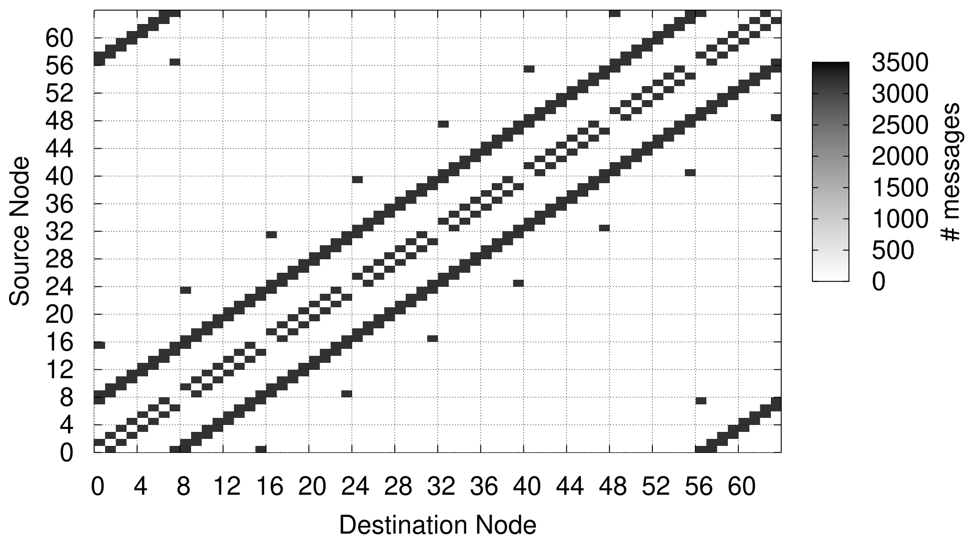

**Figure 2.** Using the data in the message density array to plot the communication pattern of the NAS SP benchmark running on 64 nodes.

model that is based on the point-to-point model is in testing for `MPI_Bcast()`. A point-to-point model is simply a function that uses the length of a message to determine how long it will take for a message of that length to traverse the network; i.e., the message delay. Seshat will take that delay into consideration at the receiving node when it updates the local virtual time.

There are point-to-point models for on-board and off-board communication. Usually it is faster to send messages between CPUs that reside on the same board, then between CPUs that must communicate over the network. Currently, Seshat assumes all communications are off-board.

The network model functions are manually created. A point-to-point ping-pong benchmark is run for increasing message lengths on the system of interest. The resulting data is then encoded as a C function inside the network model file that is compiled into Seshat. Section **??** explains how that is done.

When Seshat starts running, some network parameters can be changed. Table 3 shows the keywords that are used to do that. The function that computes the delay of a point-to-point message accepts two parameters that are used to modify the calculation. The keyword `net_lat_factor` can be used to increase or decrease the latency by a factor. The default is 1.0. The keyword `net_bw_factor` sets a multiplier for the bandwidth. Again, the default is 1.0.

The function to model collective operations also accepts a multiplier. It can be set with the

**Table 3.** Network parameters in configuration file

| **Keyword** | **Default Value** | **Description** |
|---|---|---|
| `net_bw_factor` | 1.0 | Bandwidth multiplier |
| `net_lat_factor` | 1.0 | Latency multiplier |
| `net_coll_factor` | 1.0 | Collectives multiplier |
| `net_cpu_factor` | 1.0 | Time adjustment between MPI calls |
| `net_xsection_links` | $\infty$ | Number of network links at the smalles cross section |
| `net_reporting` | no | Report networking events? |
| `mpi_long_proto` | 131072 | Cross-over point to long MPI protocol |
| `sim_sleep` | 0 | Let sim sleep that many nanoseconds at each event |

keyword `net_coll_factor`.

Seshat asumes that the MPI library of the simulated system uses eager sends for short messages, and a rendez-vous protocol for longer messages. The message length above which Seshat simulates a long message protocol can be set with the keyword `mpi_long_proto`. The default is 131,072 bytes (128 kB) when simulating a Red Storm system, 376,832 bytes (368 kB) for a Thunderbird simulation, and 1,024,000,000 bytes (1,000,000 kB) otherwise.

Seshat simulates the cross-section bandwidth of a network by assuming that all off-board traffic must cross a narrow point in the network, and limiting how much data can be in transit at any given time. For example, a simple binary tree network topology has a cross-section bandwidth that is the same bandwidth of a single link. How many links make up that cross-section can be set with the `net_xsection_links` keyword in the configuration file. The default is infinitly many, which essentially simulates a fully connected graph network.

Seshat measures the time betwen MPI calls. By default that is the native execution time of the application. To simulate faster or slower systems, the keyword `net_cpu_factor` can be set to something other than 1.0. In that case Seshat will multiply the measured time by that factor and update the virtual time the application is running in.

For debugging purposes it is possible to make Seshat sleep after each event. The number of nano seconds to sleep can be specified with the `sim_sleep` keyword.

Most of the files in Section 3.2.2 will only contain data at the end of the simulation, if the keyword `net_reporting` is set to "yes", "on", or "1".

### 3.2.4 Trace File

Two configuration file parameters can be used to enable collection of MPI trace information. The two parameters are listed in Table 4.

If the keyword `trace_file` is given, then Seshat will record an event into that file for each

**Table 4.** Trace file generation in configuration file

| **Keyword** | **Value** | **Description** |
|---|---|---|
| `trace_file` | *`file name`* | File to collect MPI trace information) |
| `msg_data_file` | *`file name`* | File to collect MPI message data) |

MPI send and MPI collective call. For each event, Seshat records the type, the virtual time, source and destination, the computed delay, the MPI tag, length in bytes, data type, and MPI count. If a message data file is specified as well, then each trace event also contains the poisition into the message data file where the data for this message begins.

The keyword `msg_data_file` can be used to specify a file that will collect the actual data sent by each MPI send and collective operation. The content of each MPI message is simply appended to that file. The starting position for each message in that file is recorded in the corresponding event in the trace file. There is one message data file per MPI rank. The name of given by the `msg_data_file` keyword plus the rank number is used to name each file.

This feature can be used to generate enough data to replay the behavior of an application. For example, it could be used to interact with another simulator that acurately runs a single rank of an MPI application. Using the data gathered with Seshat, that simulator can be fed with the messages that it would receive in a full system at the appropriate time.

Message data files can grow huge very quickly and take considerable amounts of time to write. However, under Seshat, the application does not preceive this time, since the file is written from the network simulator node. In essence, the virtual time on each compute node is paused during trace and message data file writes.

# 4 How Seshat Works

Seshat is a small library that consist of the files listed in Table 5.

## 4.1 Parameterizing the Network Model

Show how the various network parameters in the configuration file are used to parametersize the network model.

$o_\text{s}$ and $o_\text{r}$ are fixed, like LogP, but not pLogP where they are messages size $m$ dependent.

## 4.2 Core-to-Core Model

Explain the core-to-core model:

- One bus between each pair of cores
- Traffic in both directions shares the bus
- Seconding core has data ready and notifies other core: $o_\text{s}$
- Receiving core does the matching: $o_\text{r}$
- Receiving core then copies the data: $\frac{2m}{\beta_\text{bus}+\alpha_\text{bus}}$
- For (blocking) long protocol, sender has to block until receive is posted

Limitations:

- Overhead of matching is fixed (in $o_\text{r}$), does not take length of match list into consideration.
- Other bus traffic is not accounted for, although there shouldn't be much.

**Table 5.** Source files to build the Seshat library

| File name | Description |
|---|---|
| `scs.h` | The common header file. |
| `scs_config_read.c` | Rank 0 reads the configuration file and distributes the values to all nodes. |
| `scs_config_parse.c` | Parse the configuration file and extract the data |
| `scs_debug.c` | Routines to handle Seshat debugging output and functions to create and write to the output files. |
| `scs_eventQ.c` | |
| `scs_events.c` | |
| `scs_main.c` | |
| `scs_mpi_internal.c` | |
| `scs_mpi_wrappers.c` | |
| `scs_mpi_wrappers.h` | |
| `scs_mpi_wrappersC.c` | |
| `scs_mpi_wrappersF.c` | |
| `scs_mpi_wrappersF.h` | |
| `scs_mpi_wrappers_support.c` | |
| `scs_msg_data_file.c` | |
| `scs_net_capacity.c` | |
| `scs_net_main.c` | |
| `scs_net_stats.c` | |
| `scs_network_liberty.c` | The network model for Liberty, a decommissioned Myrinet cluster. |
| `scs_network_redstorm.c` | The network model for Red Storm, Sandia's Cray XT-3 running Catamount. |
| `scs_network_tbird.c` | The network model for Thunderbird, an Infiniband cluster. |
| `scs_sims.c` | |
| `scs_trace_file.c` | |
| `scs_vtime.c` | |

# References

Riesen, R. (2006a, April). **Communication Patterns**. In *Workshop on Communication Architecture for Clusters CAC'06*, Rhodes Island, Greece. IEEE.

Riesen, R. (2006b). **A Hybrid MPI Simulator**. In *IEEE International Conference on Cluster Computing (CLUSTER'06)*.

Riesen, R. (2006c). **Supercomputer Simulation Design Through Simulation**. In *Cray User Group (CUG)*.

Riesen, R. (2007a, May). **Scalable Collection of Large MPI Traces on Red Storm**. In *Cray User Group (CUG)*.

Riesen, R. (2007b). **Seshat Collects MPI Traces: Extended Abstract**. In F. Cappello, T. Herault, and J. Dongarra (Eds.), *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 14th European PVM/MPI Users' Group Meeting, Paris, France, September/October 2007 Proceedings*, Lecture Notes in Computer Science. Springer-Verlag.

Riesen, R. and T. Hoefler (2006). **Zero-Cost MPI Collectives**.

Riesen, R., C. Vaughan, and T. Hoefler (2006). **What if MPI Collective Operations Were Instantaneous?** In *Cray User Group (CUG)*.

# Index

($n$) page $n$ is in the bibliography.
[$n$] page $n$ is in the glossary.
***n*** page of a definition or a main entry.
*n* other pages where an entry is mentioned.

## DISTRIBUTION:

|  |  |  |
|---|---|---|
| 1 | MS 1319 | Rolf Riesen, 1423 |

Sandia National Laboratories