SST/macro 3.0: The Manual

Sandia National Labs
Livermore, CA

October 14, 2013

Chapter 1

Introduction

1.1 Overview

The SST/macro software package provides a simulator for large-scale parallel computer architectures. It permits the coarse-grained study of distributed-memory applications. The simulator is driven from either a trace file or skeleton application. The simulator architecture is modular, allowing it to easily be extended with additional network models, trace file formats, software services, and processor models.

Simulation can be broadly categorized as either off-line or on-line. Off-line simulators typically first run a full parallel application on a real machine, recording certain communication and computation events to a simulation trace. This event trace can then be replayed post-mortem in the simulator. Most common are MPI traces which record all MPI events, and SST/macro provides the DUMPI utility (3.9) for collecting and replaying MPI traces. Trace extrapolation can extend the usefulness of off-line simulation by estimating large or untraceable system scales without having to collect a trace, it is typically only limited to strictly weak scaling.

We turn to on-line simulation when the hardware or applications parameters need to change. On-line simulators instead run real application code, allowing native C/C++/Fortran to be compiled directly into the simulator. SST/macro intercepts certain function calls, estimating how much time passes rather than actually executing the function. In MPI programs, for example, calls to MPI_Send are linked to the simulator instead of passing to the real MPI library. If desired, SST/macro can actually be a full MPI *emulator*, delivering messages between ranks and replicating the behavior of a full MPI implementation.

Although SST/macro supports both on-line and off-line modes, on-line simulation is encouraged because event traces are much less flexible, containing a fixed sequence of events. Application inputs and number of nodes cannot be changed. Without a flexible control flow, it also cannot simulate dynamic behavior like load balancing or faults. On-line simulation can explore a much broader problem space since they evolve directly in the simulator.

For large, system-level experiments with thousands of network endpoints, high-accuracy cycle-accurate simulation is not possible, or at least not convenient. Simulation requires coarse-grained approximations to be practical. SST/macro is therefore designed for specific cost/accuracy tradeoffs. It should still capture

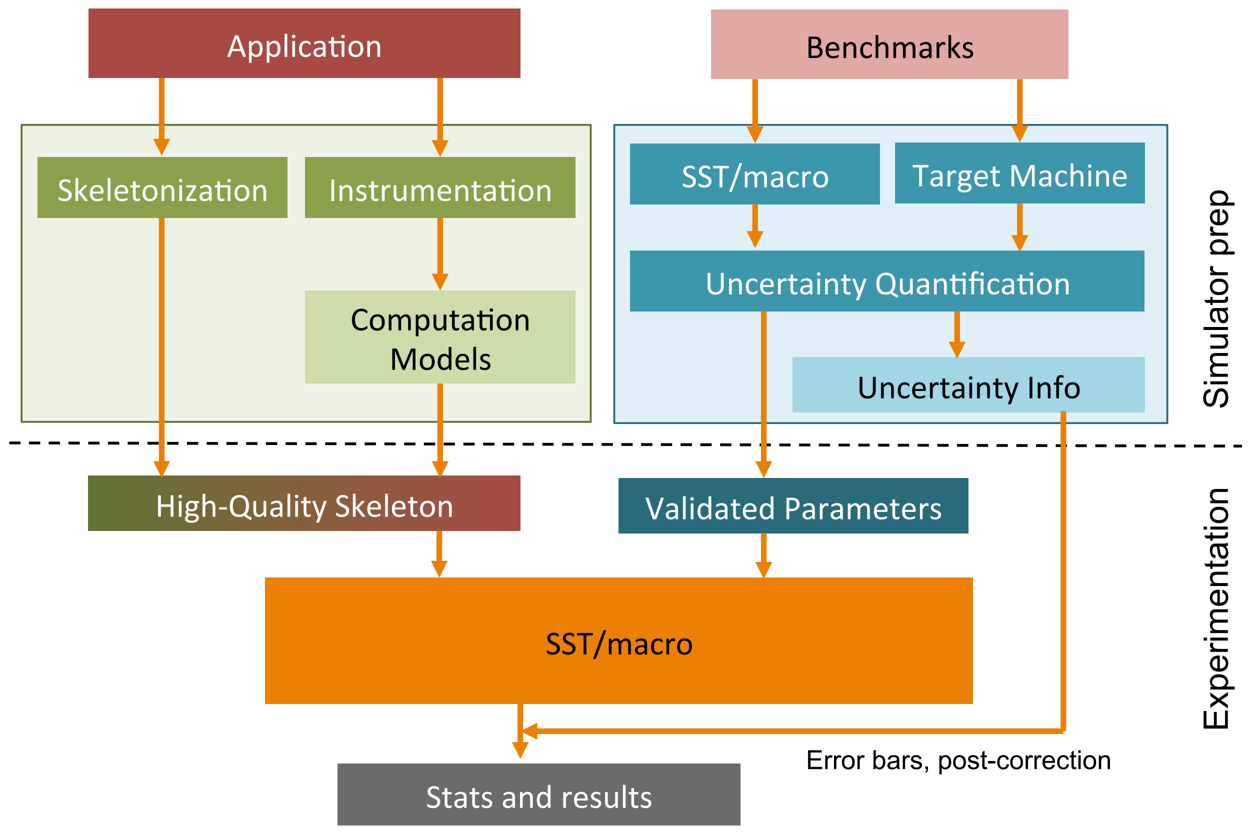


Figure 1.1: SST/macro workflow.

complex cause/effect behavior in applications and hardware, but be efficient enough to simulate at the system-level. For speeding up simulator execution, we encourage *skeletonization*, discussed further in Chapter 4. A high-quality skeleton is an application model that reproduces certain characteristics with only limited computation. We also encourage uncertainty quantification (UQ) for validating simulator results, discussed further in Chapter 5. Skeletonization and UQ are the two main elements in the "canonical" SST/macro workflow (Figure 1.1).

1.2 Currently Supported

1.2.1 Programming APIs

The following sections describe the state of the software API's (found in sstmac/software/api) that are available in SST/macro for use by applications, as of this release. The level of testing indicates the integration of compliance/functionality tests into our make check test suite.

Final and Tested

- MPI: Because of its popularity, MPI is one of our main priorities in providing programming model support. We currently test against the MPICH test suite. All tests compile, so you should never see compilation errors. However, since many of the functions are not typically used in the community, we only test commonly-used functions. See Section 1.3.2 for functions that are not supported. Functions that are not implemented will throw a sstmac::unimplementederror, reporting the function name.
- OpenSHMEM: Most of the standard OpenSHMEM tests pass. The ones that don't are because they haven't been ported to C++, or test the single unsupported feature (collect).

Some testing complete

- HPX: HPX is an implementation of the Parallex execution model. Some applications have been ported to it, and it has a simple test in the make check suite. Further development and test integration of HPX is not likely.
- Sockets: The Socket API is mostly implemented. Most basic client/server functionality is available. However, only the default socket options are allowed. In most cases, `setsockopt` is just a no-op.

In development

- Pthreads: Only the pthread_create(), pthread_join(), and pthread_self() functions are implemented. A basic pthread test validates the core spawn/run/join behavior.
- UPC: We almost have the full UPC build and runtime implemented, but no tests are currently integrated and there are many bugs to work out before it can be used.
- GNI: Cray's low-level messaging interface, GNI, is being implemented.

1.2.2 Analysis Tools and Statistics

The following analysis tools are currently available in SST/macro. Some are thoroughly tested. Others have undergone some testing, but are still considered Beta. Others have been implemented, but are relatively untested.

Fully tested

- Call graph: Generates callgrind.out file that can be visualized in either KCacheGrind or QCacheGrind. More details are given in 3.10.
- Spyplot: Generates .csv data files tabulating the number of messages and number of bytes sent between MPI ranks. SST/macro can also directly generate a PNG file. Otherwise, the .csv files can be visualized in the plotting program Scilab. More details are given in 3.11.
- Fixed-time quanta (FTQ): Generates a .csv data tabulating the amount of time spent doing computation/communication as the application progresses along with a Gnuplot script for visualization as a histogram. More details are given in 3.12

Beta

- Trace analysis: With the traceanalyzer executable, fine-grained metrics for characterizing application execution can be output.

Untested

- Congestion: With the `-d "<stats> congestion"` command line option, SST/macro will dump statistics for network congestion on individual links (packet train model only).

1.3 Known Issues and Limitations

1.3.1 Global Variables

The use of global variables in SST/macro inherently creates a false-sharing scenario because of the use of user-space threads to model parallel processes. While we do have a mechanism for supporting them (see 4.1 for more information), the file using them must be compiled with C++. This is somewhat unfortunate, because many C programs will use global variables as a convenient means of accessing program data. In almost every case, though, a C program can simply be compiled as C++ by changing the extension to .cc or .cpp.

1.3.2 MPI

Everything from MPI 2 is implemented with a few exceptions noted below. The following are *not* implemented (categorized by MPI concepts):

Communicators

- Anything using or having to do with Inter-communicators (MPI_Intercomm_create())
- Topology communicators

Datatypes and Addressing

- Complicated use of MPI_LB and MPI_UB to define a struct, and collections of structs (MPI test 138).
- Changing the name of built-in datatypes with MPI_Type_set_name() (MPI test 171).
- MPI_Create_darray(), MPI_Create_subarray(), and MPI_Create_resized()
- MPI_Pack_external() , which is only useful for sending messages across MPI implementations apparently.
- MPI_Type_match_size() - extended fortran support
- Use of MPI_BOTTOM (relative addressing). Use normal buffers.
- Using Fortran types (*e.g.*MPI_COMLEX) from C.

Info and Attributes

No MPI_Info_*, MPI_*_keyval, or MPI_Attr_* functions are supported.

Point-to-Point

- MPI_Grequest_* functions (generalized requests).
- Use of testing non-blocking functions in a loop, such as:

```
while(!flag)
{
  MPI_Iprobe( 0, 0, MPI_COMM_WORLD, &flag, &status );
}
```

  For some configurations, simulation time never advances in the MPI_Iprobe call. This causes an infinite loop that never returns to the discrete event manager. Even if configured so that time progresses, the code will work but will take a very long time to run.

Collectives

- There seems to be a problem with using MPI_FLOAT and MPI_PROD in MPI_Allreduce() (MPI test 22)
- There seems to be a problem with using non-commutative user-defined operators in MPI_Reduce() and MPI_Allreduce().
- MPI_Alltoallw() is not implemented
- MPI_Exscan() is not implemented
- MPI_Reduce_Scatter_block() is not implemented.
- MPIX_* functions are not implemented (like non-blocking collectives).
- Calling MPI functions from user-defined reduce operations (MPI test 39; including MPI_Comm_rank).

Miscellaneous

- MPI_Is_thread_main() is not implemented.

1.3.3 OpenSHMEM

Only the collect and fcollect functions of the API are not implemented (they will be in future releases).

Also, like handling of global variables discussed in Sections 1.3.1 and 4.1.1, SHMEM globals need to use a different type. For primitives and primitive arrays, refer to `<sstmac/software/api/openshmem/shmem/globals.h>` for replacing types, e.g. the following code

```
int my_global_var = 4;
double an_array[6];
```

needs to become

```
shmem_int my_global_var(4);
shmem_arr<double, 6> an_array;
```

1.3.4 Fortran

SST/macro can run Fortran90 applications. However, at least using gfortran, Fortran variables using allocate() go on the heap. Therefore, it creates a false sharing situation pretty much everywhere as threads swap in and out. A workaround is to make a big map full of data structures that store needed variables, indexed by a rank that you pass around to every function. We are exploring more user-friendly alternatives. The Fortran MPI interface is also still somewhat incomplete. Most functions are just wrappers to the C/C++ implementation and we are working on adding the bindings.

Chapter 2

Building and Running SST/macro

2.1 Build and Installation of SST/macro

2.1.1 Downloading

SST/macro is available at `http://bitbucket.org/ghendry/sstmacro`. You can get SST/macro in the following ways:

- Download a .tar of a release on the downloads page (`bitbucket.org/ghendry/sstmacro/downloads`)
- Download a .tar of the repository on the main overview page
- Clone the repository with Mercurial.

If you're using Mercurial, you can run the command:

```
$ hg clone http://bitbucket.org/ghendry/sstmacro
```

If you're behind a firewall, make sure the http proxy is set in your ~/.hgrc file:

```
[http_proxy]
host=path-to-proxy:prox-port
[https_proxy]
host=path-to-proxy:prox-port
```

If you'd like to use ssh for convenience, you'll have to modify your clone slightly by adding the "hg" username:

```
$ hg clone ssh://hg@bitbucket.org/ghendry/sstmacro
```

and also add your public key to your bitbucket user account. Also, SST/macro uses subrepos, so for using ssh you should add the following to your ~/.hgrc

```
[subpaths]
https://bitbucket.org/jpkenny/dumpi = \
  ssh://hg@bitbucket.org/jpkenny/dumpi
https://bitbucket.org/ghendry/sstmacro-pth = \
  ssh://hg@bitbucket.org/ghendry/sstmacro-pth
```

so that the http requests are converted to ssh.

2.1.2 Dependencies

- A C/C++ compiler is required. gcc 4.2 and onward is known to work.
- (optional, recommended) Qt libraries and build system (qmake) are needed to build the GUI input configuration tool. Qt 5.0 and above are suggested, although 4.9 has been observed in the wild to work (Section 2.3).
- (optional) Mercurial is needed in order to clone the source code repository, but you can also download a tar (Section 2.1.1).
- (optional, recommended) Autoconf and related tools are needed unless you are using an unmodified release or snapshot tar archive.
  - Autoconf: 2.64 or later should work and 2.68 is known to work
  - Automake: 1.11 or later should work and 1.11.1 is known to work
  - Libtool: 2.2.6 or later should work and 2.4 is known to work
- (optional) Doxygen and Graphviz are needed to build the documentation.
- (optional) Graphviz is needed to collect call graphs.
- (optional) VTK is needed for advanced vis features.
- (optional) Python with the argparse module installed is required to run UPC skeletons. Python 2.7 and on should have this.

2.1.3 Configuration and Building

For a list of known compatible systems, see Appendix B.

Once SST/macro is extracted to a directory, we recommend the following as a baseline configuration, including building outside the source tree:

```
sstmacro$ ./bootstrap.sh
sstmacro$ mkdir build
sstmacro$ cd build
sstmacro/build$ ../configure --prefix=/path-to-install
```

A complete list of options can be seen by running '../configure –help'. Some common options:

- –enable-graphviz : Enables the collection of simulated call graphs, which can be viewed with graphviz.
- –with-qt=$QMAKE: Direct SST/macro to the qmake executable. If no value is specified, it just assumes qmake is in your $PATH (see Section 2.3).
- –enable-custom-new : Memory is allocated in larger chunks in the simulator, which can speed up large simulations.
- –enable-fortran : Enable support for running fortran skeletons.

- –enable-mpiparallel: Enable parallel discrete event simulation in distributed memory over MPI. See Section 2.4.4.

Once configuration has completed, printing a summary of the things it found, simply type 'make'. It is recommended to use the '-j' option for a parallel build with as many cores as you have (otherwise it will take quite a while).

2.1.4 Post-Build

If the build did not succeed, check 2.1.5 for known issues, or contact SST/macro support for help (sstmacro-support@googlegroups.com).

If the build was successful, it is recommended to run the range of tests to make sure nothing went wrong. To do this, and also install SST/macro to the install path specified during installation, run the following commands:

```
sstmacro/build$ make -j8 check
sstmacro/build$ sudo make install
sstmacro/build$ export PATH=$PATH:/path-to-install
sstmacro/build$ make -j8 installcheck
```

Make check runs all the tests we use for development, which checks all functionality of the simulator. Make installcheck compiles some of the skeletons that come with SST/macro, linking against the installation.

Important: After SST/macro is installed, add /path-to-install/bin to your PATH variable (we keep it in our .bashrc, .profile, etc). Applications and other code linking to SST/macro use Makefiles that use the sstmacro-config script that is installed there for convenience to figure out where headers and libraries are. If you are building a skeleton (or running make installcheck) and you get errors along the lines of "can't find sstmacro-config", you probably forgot this step.

2.1.5 Known Issues

Compilation with clang should work, though one of our tests mysteriously fails so it is currently not officially supported. Hopefully soon.

2.2 Building DUMPI

By default, DUMPI is configured and built along with SST/macro with support for reading and parsing DUMPI traces, known as libundumpi. DUMPI binaries and libraries are also installed along with everything for SST/macro during make install. DUMPI can be used as it's own library within the SST/macro source tree by changing to sstmacro/dumpi, where you can change its configuration options. It is not recommended to disable libundumpi support, which wouldn't make much sense anyway.

DUMPI can also be used as stand-alone tool/library if you wish (*e.g.* for simplicity if you're only tracing). To get DUMPI by itself, either copy the sstmacro/dumpi directory somewhere else or visit `bitbucket.org/jpkenny/dumpi` and follow similar instructions for obtaining SST/macro.

To see a list of configuration options for DUMPI, run './configure –help'. If you're trying to configure DUMPI for trace collection, use '–enable-libdumpi'. Your build process might look like this (if you're building in a separate directory from the dumpi source tree) :

```
dumpi/build$ ../configure --prefix=/path-to-install --enable-libdumpi
dumpi/build$ make -j8
dumpi/build$ sudo make install
```

Warning: It is possible that the configuration process for DUMPI can take a very long time on network file systems. It basically runs through every MPI function to check its availability/status on your system. If the MPI headers and libraries are not locally available (hard drive) or cached locally, then they will be brought in each time these tests are compiled and run from storage. If storage is a parallel file system, it might be slow.

2.2.1 Known Issues

- When compiling on platforms with compiler/linker wrappers, e.g. ftn (Fortran) and CC (C++) compilers at NERSC, the libtool configuration can get corrupted. The linker flags automatically added by the wrapper produce bad values for the predeps/postdeps variable in the libtool script in the top level source folder. When this occurs, the (unfortunately) easiest way to fix this is to manually modify the libtool script. Search for predeps/postdeps and set the values to empty. This will clear all the erroneous linker flags. The compilation/linkage should still work since all necessary flags are set by the wrappers.

2.3 Building the GUI

The GUI depends on Qt 5.0 or greater. These can be easily downloaded from the Qt website. To configure SST/macro for compiling the GUI, an additional flag must be added:

```
sstmacro/build$ ../configure --prefix=/path-to-install --with-qt=$QMAKE
```

The variable $QMAKE must point to the qmake executable. If qmake is in $PATH, only '–with-qt' needs to be added. The GUI is compiled independently from SST/macro. In the build directory, just invoke:

```
sstmacro/build$ make gui
```

You will see the Qt compilation output followed by output from the source code parser. Keyword input to the GUI is automatically generated from the source code. Once parsing is complete, the GUI is ready to use. The executable is found in the qt-qui folder. On Mac, an application is generated, which can be run:

```
sstmacro/build$ open qt-gui/SSTMacro.app
```

On linux, a simple executable is generated.

2.4 Running an Application

To demonstrate how an application is run in SST/macro, we'll use a very simple send-recv program located in sstmacro/tutorials/sendrecv_c. We will take a closer look at the actual code in Section 3.3. After SST/macro has been installed and your PATH variable set correctly, run:

```
sstmacro$ cd tutorials/sendrecv_c
sstmacro/tutorials/sendrecv_c$ make
sstmacro/tutorials/sendrecv_c$ ./runsstmac -f parameters.ini
```

You should see some output that tells you 1) the estimated total (simulated) runtime of the simulation, and 2) the wall-time that it took for the simulation to run. Both of these numbers should be small since it's a trivial program.

This is how simulations generally work in SST/macro: you build skeleton code and link it with the simulator to produce a binary. Then you run that binary and pass it a parameter file which describes the machine model to use.

2.4.1 Makefiles

We recommend structuring the Makefile for your project like the one seen in tutorials/sendrecv_c/Makefile :

```
TARGET := runsstmac
SRC := $(shell ls *.c) valid_keywords.cc force_link.cc

# the sstmacro-config script must be found in PATH
CXX :=       $(shell sstmacro-config --cxx )
CC :=        $(shell sstmacro-config --cc )
CXXFLAGS := $(shell sstmacro-config --cxxflags )
CPPFLAGS := $(shell sstmacro-config --cppflags ) -I.
LIBDIR :=    $(shell sstmacro-config --libdir )
PREFIX :=    $(shell sstmacro-config --prefix )
LDFLAGS :=   $(shell sstmacro-config --ldflags ) -Wl,-rpath,$(PREFIX)/lib
...
```

The sstmacro-config script is built by the SST/macro configuration process and installed into the bin folder. More linker and include flags can be added for different source trees. For advanced usage in projects built with automate and autoconf, the sstmacro-config script can be invoked in configure.ac following the usage above.

2.4.2 C vs. C++

The three 'sendrecv' skeletons in sstmacro/tutorials show the different usage of C and C++ linking against SST/macro: C, C++ but with a C-style main, and a C++ class that inherits from sstmac::sw::mpiapp. Using C++ inheritance (such as in the sendrecv_cxx2 folder) will give you the most flexibility, including the ability to run more than one named application in a single simulation (see Section 3.2 for more info).

2.4.3 Command-line arguments

There are only a few basic command-line arguments you'll ever need to use with SST/macro, listed below

- -h - print some typical help info
- -f [parameter file] - the parameter file to use for the simulation. This can be relative to the current directory, an absolute path, or the name of a pre-set file that is in sstmacro/configurations (which installs to /path-to-install/include/configurations, and gets searched along with current directory).
- -d [debug flags] - flags that control simulator statistics and output. The general format is -d "<[category]> [flag]", where [category] can be debug, trace, or stats. For example, -d "<debug> mpi" prints out everything MPI is doing. Multiple flags can be appended with |, such as -d "<debug> mpi | <trace> dumpi". See Appendix A for a complete list of flags and what they do.
- -p [parameter]=[value] - setting a parameter value (overrides what is in the parameter file)
- -t [value] - stop the simulation at simulated time [value]
- -c - if debug output is enabled, color-code it by its source (very helpful when parsing lots of input). This uses ANSI escape sequences, so don't use this if you're producing any stats files, or redirecting output to file. Also, this is only really meant to be used on the Mac terminal.
- -r [run number] - for a parameter file that is enabled for a parameter sweep, run only a specific parameter combination. See Section 3.2.1.

2.4.4 Parallel Simulation (Beta)

SST/macro supports running a parallel discrete event simulation (PDES) in distributed memory over MPI. First, you must configure the simulator with the '–enable-mpiparallel' flag. Configure will check for MPI, and that you're using the standard MPI compiler wrappers to build all of SST/macro, so that a consistent compiler is being used everywhere. So your configure should look something like this:

```
sstmacro/build $ ../configure --enable-mpiparallel CXX=mpicxx CC=mpicc ...
```

SST/macro also uses METIS to partition the network topology, so make sure you have that installed somewhere and the binary is in your PATH. At the time of this writing, METIS can be found at `http://glaros.dtc.umn.edu/gkhome/metis/metis/download`. Next, you must use the '–ldflagsparallel' option of the sstmacro-config script in the Makefile that you're using to link SST/macro into your code, like so:

```
...
LDFLAGS :=  $(shell sstmacro-config --ldflagsparallel ) ...
...
```

which links in the correct parts of the simulator for running parallel simulations. Finally, use your favorite MPI launcher to run SST/macro like normal:

```
mysim $ mpirun -n 4 ./runsstmac -f parameters.ini -d ''<debug> mpicheck"
```

SST/macro will automatically start up, figure out how many partitions (MPI processes) you're using, partition the network topology into contiguous blocks, and start running in parallel. Note that you should always use the 'mpicheck' debug flag and use what it reports as the final simulation time (the simulation time that SST/macro reports by default will be different across all partitions. mpicheck assures a consistent time is reported).

Parallel simulation may not (and probably won't) speed up SST/macro. Most events are scheduled farther into the future than link (synchronization) latency. Since we use the conservative null-message technique, there will be a lot of overhead in synchronizing the clocks. Parallel is most likely to be useful because of memory constraints. In this case, we recommend limiting parallel simulation to as few nodes as possible (ideally, with as much shared memory as possible).

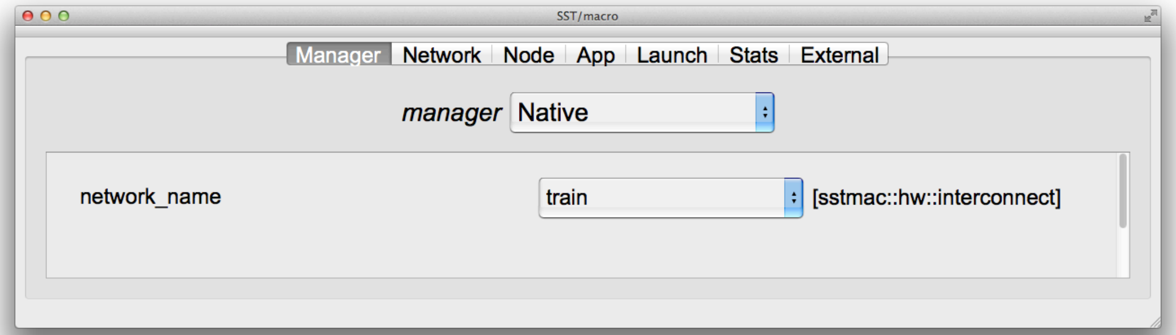
Chapter 3

Basic Tutorials

3.1 SST/macro GUI

The SST/macro GUI is used for creating parameter files for an SST/macro simulation, keeping track of which parameters are required depending on the different models available. See Section 2.3 for instructions on building and running the GUI.

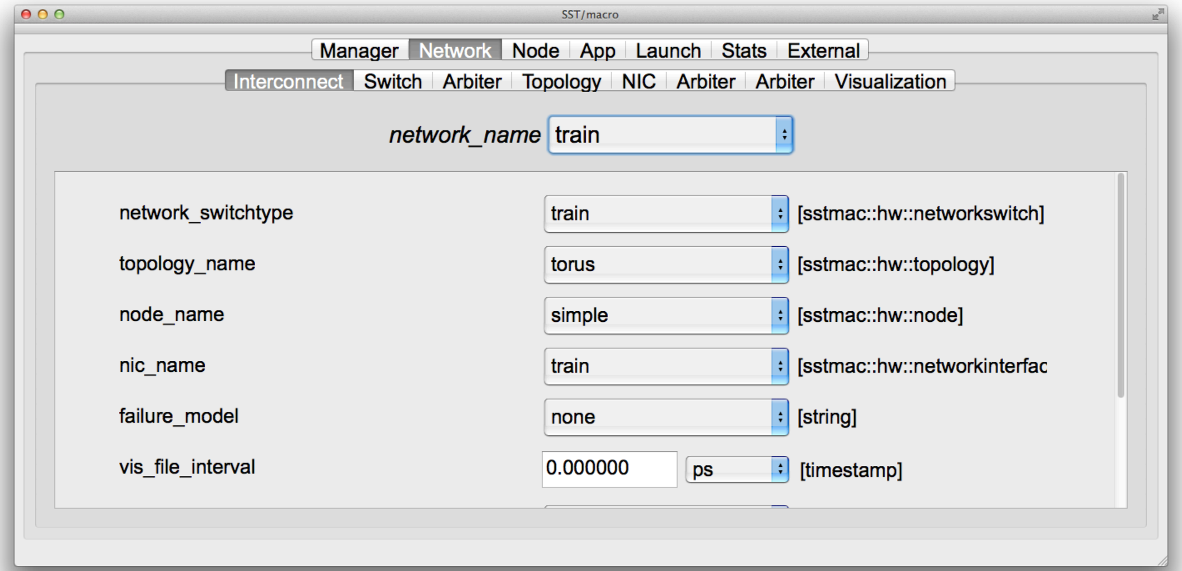
The GUI displays the various SST/macro components and sub-components as separate tabs.



manager Native

network_name train [sstmac::hw::interconnect]

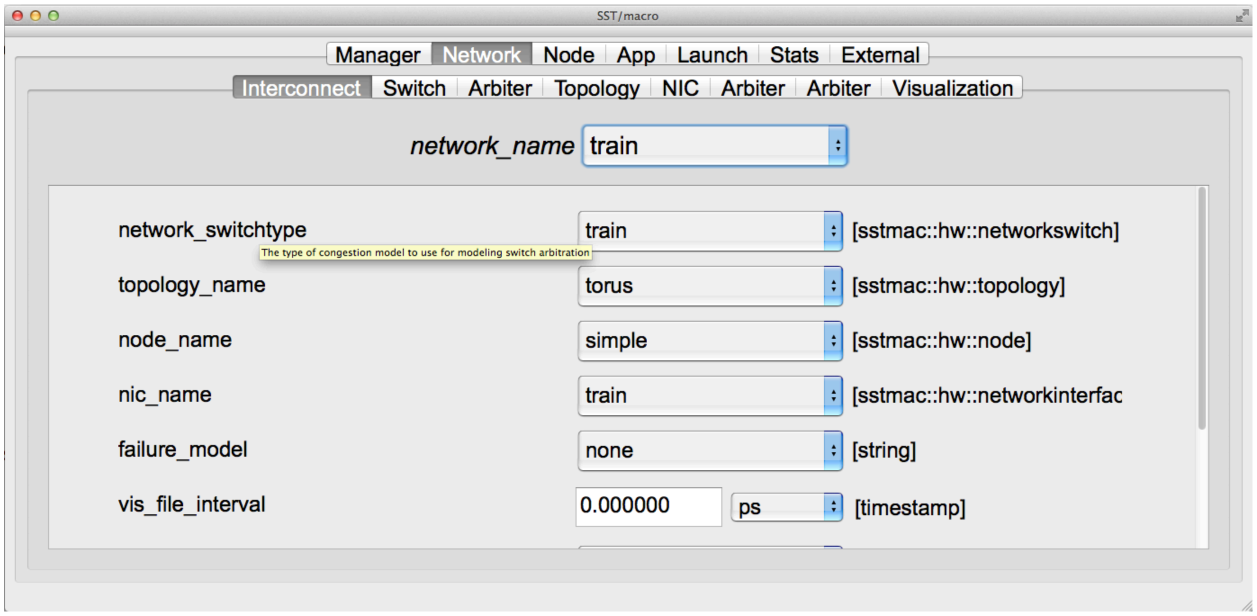
The opening tab shows the discrete event manager. On the top, tabs for the major components can be seen such as the network, node, and the application. On the network tab, we have several subtabs for the various network components.



network_name train

network_switchtype	train	[sstmac::hw::networkswitch]
topology_name	torus	[sstmac::hw::topology]
node_name	simple	[sstmac::hw::node]
nic_name	train	[sstmac::hw::networkinterfac
failure_model	none	[string]
vis_file_interval	0.000000 ps	[timestamp]

Here the most important tabs are interconnect, switch, topology, and NIC. We see the keyword *network_switchtype* with value *train*. Most keywords are documented with tooltips to give a brief introduction to what each keyword means.



network_name train

network_switchtype	train	[sstmac::hw::networkswitch]
topology_name	torus	[sstmac::hw::topology]
node_name	simple	[sstmac::hw::node]
nic_name	train	[sstmac::hw::networkinterfac
failure_model	none	[string]
vis_file_interval	0.000000 ps	[timestamp]

The type of congestion model to use for modeling switch arbitration

The tooltip indicates this keyword determines which model to use for simulating congestion in the network switches. The exact meaning of *train* is beyond the scope of this section (and the tooltip). For a brief introduction to congestion model, see 3.7.

At present, the GUI is only intended as a helper tool for constructing input files. Once a configuration is chosen, a parameter file can be generated by selecting File → Save and creating a *.ini* file. Although a more comprehensive GUI is planned, experiments must still be run on the command line via parameter files.

3.2 SST/macro Parameter files

A minimal parameter file setting up a 2D-torus topology is shown below:

```
# Launch parameters
launch_name = instant
launch_indexing = block
launch_allocation = firstavailable
launch_app1_cmd = aprun -n2 -N1
launch_app1 = user_mpiapp_cxx
launch_app1_argv =
# Network parameters
network_name = analytic
network_bandwidth = 1.0GB/s
network_latency = 2us
# Topology - Ring of 4 nodes
topology_name = hdtorus
topology_geometry = 4,4
# Node parameters
node_cores = 1
node_name = null
node_memory_model = null
nic_name = null
# Application parameters
sendrecv_message_size = 128
```

The input file follows a basic syntax of `parameter = value`. Parameter names follow C++ variable rules (letters, numbers, underscore) while parameter values can contain spaces. Trailing and leading whitespaces are stripped from parameters. Comments can be included on lines starting with #.

The input file is broken into sections via comments. First, application launch parameters must be chosen determining what application will launch, how nodes will be allocated, how ranks will be indexed, and finally what application will be run. Additionally, you must specify how many processes to launch and how many to spawn per node. We currently recommend using aprun syntax (the launcher for Cray machines), although support is being added for other process management systems. SST/macro can simulate command line parameters by giving a value for `launch_app1_argv`.

A network must also be chosen. In the simplest possible case, the network is modeled via a simple latency/bandwidth formula. For more complicated network models, many more than two parameters will be required. See 3.7 for a brief explanation of SST/macro network congestion models. A topology is also needed for constructing the network. In this case we choose a 2-D 4×4 torus (16 switches). The `topology_geometry` parameter takes an arbitrarily long list of numbers as the dimensions to the torus.

Finally, we must construct a node model. In this case, again, we use the simplest possible models (null model) for the node, network interface controller (NIC), and memory. The null model is essentially a no-op, generating the correct control flow but not actually simulating any computation. This is useful for validating program correctness or examining questions only related to the network. More accurate (and complicated) models will require parameters for node frequency, memory bandwidth, injection latency, etc.

Parameter files can be constructed in a more modular way through the `include` statement. An alternative parameter file would be:

```
include machine.ini
# Launch parameters
launch_name = instant
launch_indexing = block
launch_allocation = firstavailable
launch_app1_cmd = aprun -n2 -N1
launch_app1 = user_mpiapp_cxx
launch_app1_argv =
# Application parameters
sendrecv_message_size = 128
```

where in the first line we include the file `machine.ini`. All network, topology, and node parameters would be placed into a `machine.ini` file. In this way, multiple experiments can be linked to a common machine. Alternatively, multiple machines could be linked to the same application by creating and including an `application.ini`.

3.2.1 Parameter Sweeping with fork()

Often, an experiment consists of sweeping one or more parameters and observing the effect on performance. This can be accomplished by specifying a set of parameter values in brackets, separated by |, like so:

```
...
sendrecv_message_size = {128|256|512|1024}
multisim_nproc = 2
```

SST/macro will automatically fork to run each configuration separately, and a file for each configuration will be created with the simulator output. The `multisim_nproc` parameter controls how many simulations will be run concurrently. If multiple parameters specify ranges, then SST/macro will run every permutation of them. Sometimes only a subset of all permutations is desired, so to address this you can also apply this mechanism to the `include` directive, like so:

```
include {machine1.ini | machine2.ini}
...
sendrecv_message_size = {128|256|512|1024}
multisim_nproc = 10
```

Note that an *included* parameter file can also have parameter ranges, but this can get out of hand quickly keeping track of everything, so we recommend sticking to the above methods.

The simulator will number each configuration and print this numbering before it starts anything, and ask you to hit enter to continue. If you want to run a single configuration, usually for debugging without having to change the parameter file, you can specify the -r [number] command line option to run just that configuration.

3.3 Basic MPI Program

Let us go back to the simple send/recv skeleton and actually look at the code.

```
#include <stdlib.h>
#include <stdio.h>
#include <sstmac/sstmpi.h>
#include <sstmac/util.h>

int user_skeleton_main(int argc, char **argv)
{
    int message_size = 128;
    int me, nproc;
    int tag = 0;
    int dst = 1;
    int src = 0;
    MPI_Status stat;

    MPI_Init(&argc,&argv);
    MPI_Comm world = MPI_COMM_WORLD;
    MPI_Comm_rank(world,&me);
    MPI_Comm_size(world,&nproc);
```

The starting point is creating a main routine for the application, which should be named `user_skeleton_main`. When SST/macro launches, it will invoke this routine and pass in any command line arguments specified via the `launch_app1_argv` parameter. Upon entering the main routine, the code is now indistinguishable from regular MPI C code. In the parameter file to be used with the simulation, you must set

```
launch_app1 = user_mpiapp_c
```

or, if compiling as C++

```
launch_app1 = user_mpiapp_cxx
```

At the very top of the file, the SST/macro header files must be included. This header provides the MPI API and configures MPI function calls to link to SST/macro instead of the real MPI library. In most cases, only `sstmpi.h` will be needed, but `util.h` contains utility functions that may be useful. If trying to maintain code as "single-source" compilable without modification for both SST/macro and actual MPI, we recommend creating a `parallel.h`.

```
#ifdef SSTMACRO
#include <sstmac/sstmpi.h>
#include <sstmac/util.h>
#define USER_MAIN user_skeleton_main
#else
#include <mpi.h>
#define USER_MAIN main
#endif
```

This creates a central point for swapping back and forth. The beginning of the file would then become

```
#include <stdlib.h>
#include <stdio.h>
#include <parallel.h>

int USER_MAIN(int argc, char **argv)
{
```

The end of the file is again just regular MPI C code:

```
    if (nproc != 2) {
        fprintf(stderr, "sendrecv only runs with two processors\n");
        abort();
    }
    if (me == 0) {
        MPI_Send(NULL, message_size, MPI_INT, dst, tag, world);
        printf("rank %i sending a message\n", me);
    }
    else {
        MPI_Recv(NULL, message_size, MPI_INT, src, tag, world, &stat);
        printf("rank %i receiving a message\n", me);
    }
    MPI_Finalize();
    return 0;
}
```

Here the code just checks the MPI rank and sends (rank 0) or receives (rank 1) a message.

3.4 Simple Message API

For more advanced usage, users may wish to code more directly with SST/macro C++ functions. If coding directly in MPI, you can only send messages as `void*` arrays. Additionally, MPI only exposes basic send/recv functionality leaving little control over the exact protocol. If using SST/macro to design a code BEFORE the code actually exists, the recommended path is to use the SST/macro `simp` API (simple messages). Send/recv calls pass C++ objects rather than `void*` arrays, allowing data structures to be directly passed between parallel processes. Additionally, simple functions for datagram send, RDMA, and polling are available, allowing the user to experiment directly with HOW a message is sent. The added flexibility of the C++ interface is potentially beneficial for exploring performance tradeoffs when laying out the initial design of a project. In general, the `simp` API is geared towards asynchronous execution models rather than bulk-synchronous.

3.4.1 Initial Setup

The code for the example can be found in `skeletons/simpmsg`. To begin, the basic header files must be included.

```
#include <sstmac/simpmsg.h>

using namespace sstmac;
using namespace sstmac::sw;
using namespace sstmac::hw;

#define debug_print(...) std::cout << sstprintf(__VA_ARGS__)

namespace simpmsg {

sstmac_register_app(simpmsg);
```

To simplify the use of SST/macro objects, we declare the three major SST/macro namespaces. For debug printing, SST/macro provides a utility function, `sstprintf`, that functions exactly like `printf` but is compatible with C++ output streams (`std::cout`). To demonstrate slightly more advanced usage, we declare a new application named "simpmsg" using the `sstmac_register_app` macro.

To create and send messages, we need to create a message class.

```
class test_message :
    public simple_message
{
...
};
```

We must inherit form the `simple_message` type. Beyond that, the class can be structured however the user wants. In this test case, each message will perform an action for some value.

```
    public:
        typedef enum { compute, request, terminate, data } action_t;

    private:
        /**
        *    Number of bytes requested OR
        *    Number of bytes being sent OR
        *    Number of microseconds to compute
        */
        int value_;

        action_t action_;
```

The meaning of `value` depends on the action, being either the number of bytes to send or the number of μs to compute.

SST/macro is built on top of the Boost smart pointers header library for memory management. To simplify reading of the code and shorten type names, we explicitly typedef all pointer types in the class.

```
    public:
        typedef boost::intrusive_ptr<test_message> ptr;
```

The next part of the code (omitted) implements various constructors. The most important function call to implement is the `clone` method

```
        sst_message::ptr
        clone(MESSAGE_TYPES ty) const
        {
            test_message::ptr cln = new test_message(value_, action_);
            simple_message::clone_into(ty, cln);
            simple_message::clone_into(cln);
            return cln;
        }
```

which overrides a virtual function in the parent class. Messages must be cloned for various subtle reasons through the code. To ensure that the `simple_message` interface is cloned properly, function calls must be made to the parent `clone_into` method.

Now that we have a class for sending messages, we can create a thread that will poll for incoming messages.

```
class messenger_thread :
    public simpmsg_thread
{
    private:
        simpmsg_queue::ptr work_queue_;

    public:
        typedef boost::intrusive_ptr<messenger_thread> ptr;

        messenger_thread(const simpmsg_queue::ptr& queue)
            : work_queue_(queue)
        {
        }

        virtual void run();

        void terminate();
};
```

As before, the thread object must inherit from a simple message type. Additionally, we will make use of a another type: `simpmsg_queue` that can be used for managing work queues between threads. Every thread must implement a run method, which will be invoked by the SST/macro operating system.

```
void
messenger_thread::run()
{
    int me = simp_rank();
    while(1)
    {
        simple_message::ptr msg = simp_poll();
        test_message::ptr tmsg = safe_cast(msg, test_message);
        debug_print("receiver got message of type %s on %d\n",
                    test_message::tostr(tmsg->action()), me);
        work_queue_->put_message(msg);
        if (tmsg->action() == test_message::terminate)
            break;
    }
}
```

In this case, the loop just runs continuously polling for messages. Because of subtleties in discrete event simulation, `simp_poll` blocks until a message is found. As mentioned in the introduction, non-blocking probe/polls are not really amenable to discrete event simulation. The messenger thread performs no real work and simply places the message received into a work queue.

Now we can skip ahead to the actual `main` routine.

```
int
simpmsg_main(int argc, char** argv)
{
    simp_init();
    int me = simp_rank();
    int nproc = simp_nproc();
    std::cout << sstprintf("Rank %d starting\n", me);

    //spin off a messenger thread
    simpmsg_queue::ptr work_queue = simpmsg_queue::construct();
    messenger_thread::ptr thr = new messenger_thread(work_queue);
    thr->start();

    run_work_loop(me, nproc, work_queue);

    thr->terminate();
    thr->join();

    std::cout << sstprintf("Rank %d terminating\n", me);
    simp_finalize();
    return 0;
}
```

As in MPI, we must call initialize/finalize routines at the beginning and end of main. Additionally, each process is still assigned a process id (rank) and told the total number of ranks. The code first creates a work queue, creates a messenger thread based on that work queue, and then starts the thread running. The start method invokes all the necessary SST/macro operating system routines, shielding the user from details of the discrete event scheduler. Once the messenger thread is running, the main thread enters a work loop.

```
void
run_work_loop(int me, int nproc, const simpmsg_queue::ptr& work_queue)
{
    // send out some data requests and some tasks
    for (int t=1; t <= max_num_tasks; ++t){
        int dst = (me + t) % nproc;
        simp_send(dst, new test_message(1e4, test_message::compute));
        simp_send(dst, new test_message(1e6, test_message::request));
    }
```

The work loop begins by sending out (in round-robin fashion) compute tasks that take $10^4 \mu s$. Additionally, it sends out data requests for messages of size 1 MB. Once all the tasks and data requests have been sent, the worker thread starts pulling work from the queue.

```
    while (1)
    {
        simple_message::ptr msg = work_queue->poll_until_message();
        test_message::ptr tmsg = safe_cast(msg, test_message);
        debug_print("got work message of type %s on %d\n",
                    test_message::tostr(tmsg->action()), me);
        switch (tmsg->action())
        {
          ...
        }
```

Depending on the type of message received, the worker thread will perform various actions. If a compute message is received:

```
            case test_message::compute:
                //compute for x microseconds
                compute(tmsg->value()*1e-6);
                ++num_tasks;
                if (terminate(num_tasks, num_sends))
                    return;
                break;
```

The function `compute` is included by the header files and simulates a thread computing for a given number of seconds (hence the factor of 10^{-6}). The code tracks the number of tasks and number of send requests received. The function `terminate` checks to see if all the tasks and sends have been completed.

```
            case test_message::compute:
                //compute for x microseconds
                compute(tmsg->value()*1e-6);
                ++num_tasks;
                if (terminate(num_tasks, num_sends))
                    return;
                break;
```

If a request message is received

```
            case test_message::request:
                //send back x bytes of data
                simp_rdma_put(tmsg->src(), new test_message(tmsg->value()));
                break;
```

the code responds to the request by performing an RDMA put of `tmsg->value()` number of bytes to the requesting node. If an RDMA put completes at the destination, an ack is generated to signal the node that data is now available.

```
            case test_message::rdma_put_ack:
                ++num_sends;
                if (terminate(num_tasks, num_sends))
                    return;
                break;
```

Finally, a terminate message might be sent, causing the thread to quit the work loop.

```
            case test_message::terminate:
                return;
```

This example shows basic usage for a very simple asynchronous execution model. Although the code cannot be compiled and actually run on an HPC machine the way an MPI C code can, the C++ interface provides greater flexibility and potentially makes performance experiments easier.

3.5 Network Topologies and Routing

We here give a brief introduction to specifying different topologies and routing strategies. We will only discuss one basic example (torus). A more thorough introduction covering all topologies is planned for future releases. Excellent resources are "Principles and Practices of Interconnection Networks" by Brian Towles and William Dally published by Morgan Kaufman and "High Performance Datacenter Networks" by Dennis Abts and John Kim published by Morgan and Claypool.

3.5.1 Topology

Topologies are determined by two mandatory parameters.

```
topology_name = torus
topology_geometry = 4 4
```

Here we choose a 2D-torus topology with extent 4 in both the X and Y dimensions for a total of 16 nodes (Figure 3.1) The topology is laid out in a regular grid with network links connecting nearest neighbors. Additionally, wrap-around links connect the nodes on each boundary.

The figure is actually an oversimplification. The `topology_geometry` parameter actually specifies the topology of the *network switches*, not the compute nodes. A torus is an example of a direct network in which each switch has one or more nodes "directly" connected to it. A more accurate picture of the network is given in Figure 3.2. While in many previous architectures there was generally a one-to-one correspondence between compute nodes and switches, more recent architectures have multiple compute nodes per switch (e.g. Cray Gemini with two nodes). Multinode switches can be specified via

```
topology_name = torus
topology_geometry = 4 4
network_nodes_per_switch = 2
```

which would now generate a torus topology with 16 switches and 32 compute nodes.

Another subtle modification of torus (and other networks) can be controlled by giving the X, Y, and Z directions different bandwidth. The above network could be modified as

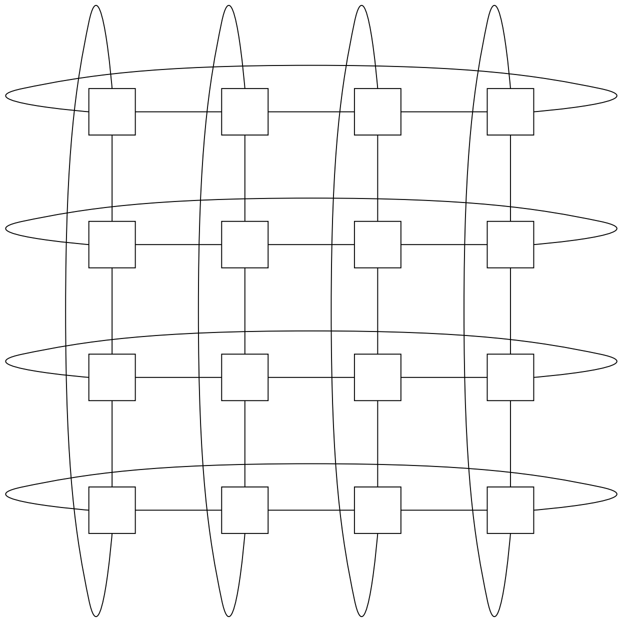


Figure 3.1: 4 x 4 2D Torus

```
topology_name = torus
topology_geometry = 4 4
topology_redundant = 2 1
```

giving the the X-dimension twice the bandwidth of the Y-dimension. This pattern DOES exist in some interconnects as a load-balancing strategy. A very subtle point arises here. Consider two different networks:

```
topology_name = torus
topology_geometry = 4 4
topology_redundant = 1 1
network_bandwidth = 2GB/s
```

```
topology_name = torus
topology_geometry = 4 4
topology_redundant = 2 2
network_bandwidth = 1GB/s
```

For some coarse-grained models, these two networks are exactly equivalent. In more fine-grained models, however, these are actually two different networks. The first network has ONE link carrying 2 GB/s. The second network has TWO links each carrying 1 GB/s.

3.5.2 Routing

By default, SST/macro uses the simplest possible routing algorithm: dimension-order minimal routing (Figure 3.3). In going from source to destination, the message first travels along the X-dimension and then travels

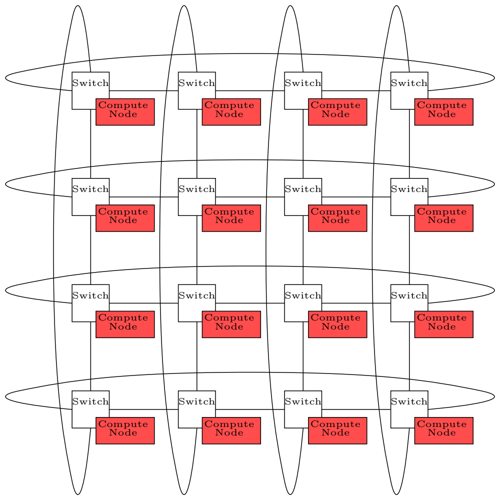


Figure 3.2: 4 x 4 2D Torus of Network Switches with Compute Nodes

along the Y-dimension. The above scheme is entirely static, making no adjustments to avoid congestion in the network. SST/macro supports a variety of adaptive routing algorithms. This can be specified:

```
default_routing = min_ad
```

which specifies minimal adaptive routing. There are now multiple valid paths between network endpoints, one of which is illustrated in Figure 3.4. At each network hop, the router chooses the *productive* path with least congestion. In some cases, however, there is only one minimal path (node $(0, 0)$ sending to $(2, 0)$ with only X different). For these messages, minimal adaptive is exactly equivalent to dimension-order routing. Other supported routing schemes are valiant and UGAL. More routing schemes are scheduled to be added in future versions. A full description of more complicated routing schemes will be given in its own chapter in future versions. For now, we direct users to existing resources such as "High Performance Datacenter Networks" by Dennis Abts and John Kim.

3.6 Discrete Event Simulation

Although not necessary for using the simulator, a basic understanding of discrete event simulation can be helpful in giving users an intuition for network models and parameters. Here we walk through a basic program that executes a single send/recv pair. SST/macro simulates many parallel processes, but itself runs as a single process with only one address space (SST/macro can actually run in parallel mode, but we ignore that complication here). SST/macro manages each parallel process as a user-space thread (application thread), allocating a thread stack and frame of execution. User-space threading is necessary for large simulations since otherwise the kernel would be overwhelmed scheduling thousands of threads.

SST/macro is driven by a simulation thread which manages the user-space thread scheduling (Figure 3.5). In the most common (and simplest) use case, all user-space threads are serialized, running one at a time.

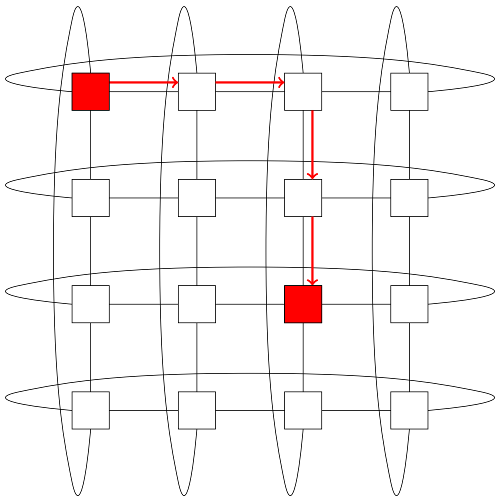


Figure 3.3: Dimension-Order Minimal Routing on a 2D Torus

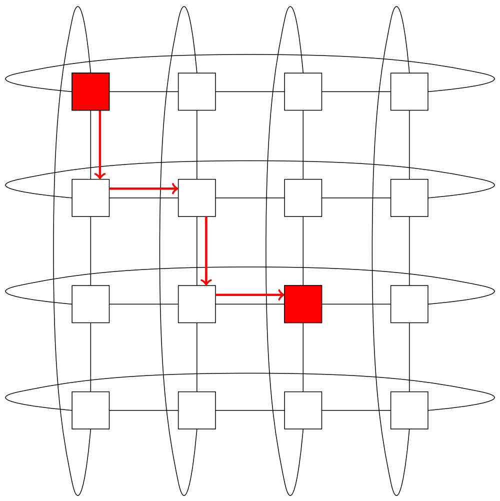
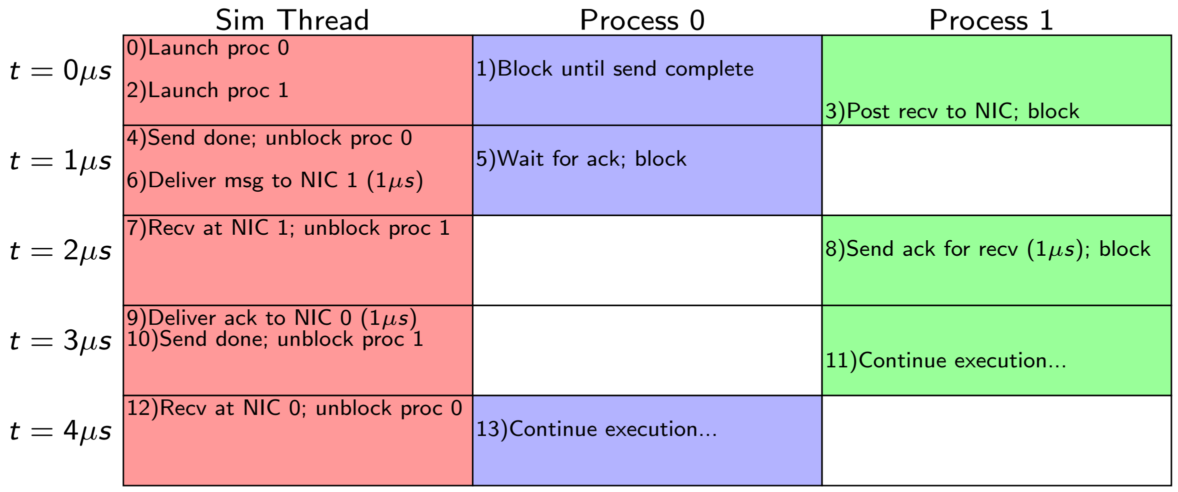


Figure 3.4: Adaptive Minimal Routing on a 2D Torus

The main simulation thread must manage all synchronizations, yielding execution to process threads at the appropriate times. The main simulation thread is usually abbreviated as the DES (discrete event simulation) thread. The simulation progresses by scheduling future events. For example, if a message is estimated to take 5 μs to arrive, the simulator will schedule a MESSAGE ARRIVED event 5 μs ahead of the current time stamp. Every simulation starts by scheduling the same set of events: launch process 0, launch process 1, etc.

The simulation begins at time $t = 0\mu s$. The simulation thread runs the first event, launching process 0. The context of process 0 is switched in, and SST/macro proceeds running code as if it were actually process



	Sim Thread	Process 0	Process 1
$t = 0\mu s$	0)Launch proc 0 2)Launch proc 1	1)Block until send complete	3)Post recv to NIC; block
$t = 1\mu s$	4)Send done; unblock proc 0 6)Deliver msg to NIC 1 ($1\mu s$)	5)Wait for ack; block	
$t = 2\mu s$	7)Recv at NIC 1; unblock proc 1		8)Send ack for recv ($1\mu s$); block
$t = 3\mu s$	9)Deliver ack to NIC 0 ($1\mu s$) 10)Send done; unblock proc 1		11)Continue execution...
$t = 4\mu s$	12)Recv at NIC 0; unblock proc 0	13)Continue execution...	

Figure 3.5: Progression of Discrete Event Simulation for Simple Send/Recv Example

0. Process 0 starts a blocking send in Event 1. For process 0 to perform a send in the simulator, it must *schedule* the necessary events to simulate the send. Most users of SST/macro will never need to explicitly schedule events. Discrete event details are always hidden by the API and executed inside library functions. In this simple case, the simulator estimates the blocking send will take 1 μs. It therefore schedules a SEND DONE (Event 4) 1 μs into the future before blocking. When process 0 blocks, it yields execution back to the main simulation.

At this point, no time has yet progressed in the simulator. The DES thread runs the next event, launching process 1, which executes a blocking receive (Event 3). Unlike the blocking send case, the blocking receive does not schedule any events. It cannot know when the message will arrive and therefore blocks without scheduling a RECV DONE event. Process 1 just registers the receive and yields back to the DES thread.

At this point, the simulator has no events left at t=0 μs and so it must progress its time stamp. The next event (Event 4) is SEND DONE at t=1 μs. The event does two things. First, now that the message has been injected into the network, the simulator estimates when it will arrive at the NIC of process 1. In this case, it estimates 1 μs and therefore schedules a MESSAGE ARRIVED event in the future at t=2 μs (Event 7). Second, the DES thread unblocks process 0, resuming execution of its thread context. Process 0 now posts a blocking receive, waiting for process 1 to acknowledge receipt of its message.

The simulator is now out of events at t=1 μs and therefore progresses its time stamp to t=2 μs. The message arrives (Event 7), allowing process 1 to complete its receive and unblock. The DES thread yields execution back to process 1, which now executes a blocking send to ack receipt of the message. It therefore schedules a SEND DONE event 1 μs in the future (Event 10) and blocks, yielding back to the DES thread. This flow of events continues until all the application threads have terminated. The DES thread will run out of events, bringing the simulation to an end.

3.7 Network Model

3.7.1 Packet

The packet model is the simplest and most intuitive of the congestion models for simulating network traffic. The physics correspond naturally to a real machine: messages are broken into small chunks (packets) and routed individually through the network. When two messages compete for the same channel (Figure 3.6A), arbitration occurs at regular intervals to select which packet has access. Packets that lose arbitration are delayed, leading to network congestion. In SST/macro, the packet model is still *coarse-grained*. In a real machine, packet sizes can be very small (100 B). Additionally, arbitration can happen on flits (flow control units), an even smaller unit than the packet. Flit-level arbitration or even 100B packet arbitration is far too fine-grained to do system-level simulation. While packet size is tunable in SST/macro, the simulator is designed for coarse-grained packet sizes of 1 KB to 8 KB. The same flow control (routing, arbitration, congestion avoidance) is performed on coarse-grained packets, but some accuracy is lost.

The coarse-grained packet model has two main sources of error. First, coarse-grained packets systematically overestimate (de)serialization latency. Before a packet can be forwarded to its next destination, it must be completely deserialized off the network link into a buffer. In a real machine, data can be forwarded on a flit-by-flit basis, efficiently pipelining packets. Flits that would send in a real system are artificially delayed until the rest of the coarse-grained packet arrives. Second, coarse-grained packets exclusively reserve network links for the entire length of the packet. In a real machine, two packets could multiplex across a link on a flit-by-flit basis.

For more details on packet parameters, see the `hopper.ini` file in the `configurations` folder in the SST/macro source. The packet model can also be explored via the GUI (see 3.1).

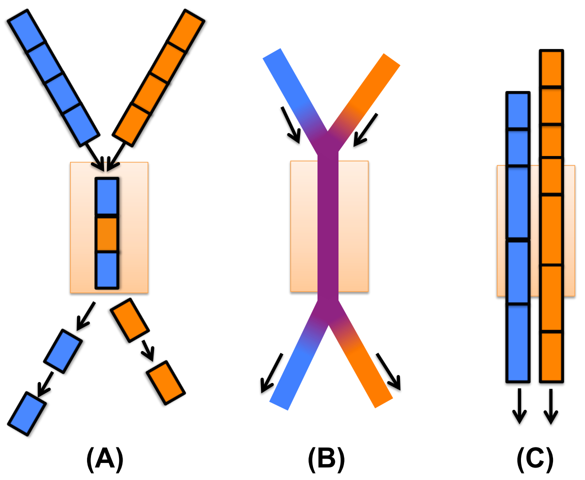


Figure 3.6: Schematic of two messages competing for same channel in the different SST/macro congestion models. (A) Packet Model (B) Flow Model (C) Train Model

3.7.2 Flow

The flow model, in simple cases, corrects the most severe problems of the packet model. Instead of discrete chunks, messages are modeled as fluid flows moving through the network (Figure 3.6B). Congestion is treated as a fluid dynamics problem, sharing bandwidth between competing flows. Without congestion, a flow only requires a FLOW START and FLOW STOP event to be modeled (see tutorial on discrete event simulation in 3.6). While the packet model would require many, many events to simulate a 1 MB message, the flow model might only require two. With congestion, flow update events must be scheduled whenever congestion changes on a network link. For limited congestion, only a few update events must occur. The flow model also corrects the latency and multiplexing problems in the packet model, providing higher-accuracy for coarse-grained simulation.

The flow model starts to break down for large systems or under heavy congestion. In the packet model, all congestion events are "local" to a given router. The number of events is also constant in packet models regardless of congestion since we are modeling a fixed number of discrete units. In flow models, flow update events can be "non-local," propagating across the system and causing flow update events on other routers. When congestion occurs, this "ripple effect" can cause the number of events to explode, overwhelming the simulator. For large systems or heavy congestion, the flow model is actually much slower than the packet model.

SST/macro actually implements a modified "fast-flower" model that introduces new approximations to avoid the ripple effect. With modest congestion, the approximations are still relatively accurate and should produce good results. For more details on flow parameters, see the `hopper_flow.ini` file in the `configurations` folder in the SST/macro source. The flow model can also be explored via the GUI (see 3.1).

3.7.3 Train

Train is a hybrid-model of flow and packet, trying to correct the latency errors in the packet model while avoiding the ripple effect of the flow model. The term train is applied since each train is a medium-size unit (1-8 KB) of many packets (100 B) linked together and traveling the same speed on the same route. Much like packets, the train model begins by converting messages into many discrete chunks of fixed size. In contrast to the packet model, channel arbitration is not exclusive. When multiple trains compete for a channel, each train "samples" the current congestion (Figure 3.6C). Based on congestion, the train estimates its bandwidth and latency. If low congestion is sampled, high-bandwidth is assigned (short packet in the figure). If high congestion is sampled, low-bandwidth is assigned (long packet in the figure).

The train model corrects the two most important packet model errors. Once bandwidth has been assigned, the packet can immediately be forwarded to the next router, producing accurate latencies. The sampling procedure also allows two packets to multiplex across a channel. Because messages are broken into discrete chunks, the number of events per message is constant regardless of congestion, avoiding the ripple effect.

For more details on train parameters, see the `hopper_train.ini` file in the `configurations` folder in the SST/macro source. The train model can also be explored via the GUI (see 3.1). The train model is currently recommended and is the default configuration when the GUI is opened.

3.8 Launching, Allocation, and Indexing

3.8.1 Launch Commands

Just as jobs must be launched on a shared supercomputer using Slurm or aprun, SST/macro requires the user to specify a launch command for the application. Currently, we encourage the user to use aprun from Cray, for which documentation can easily be found online. In the parameter file you specify, e.g.

```
launch_app1 = user_mpiapp_cxx
launch_app1_cmd = aprun -n 8 -N 2
```

which launches an external user C++ application with eight ranks and two ranks per node. The aprun command has many command line options (see online documentation), some of which may be supported in future versions of SST/macro. In particular, we are in the process of adding support for thread affinity, OpenMP thread allocation, and NUMA containment flags. Most flags, if included, will simply be ignored.

3.8.2 Allocation Schemes

In order for a job to launch, it must first allocate nodes to run on. Here we choose a simple 2D torus

```
topology_name = torus
topology_geometry = 3 3
network_nodes_per_switch = 1
```

which has 9 nodes arranged in a 3x3 mesh. For the launch command `aprun -n 8 -N 2`, we must allocate 4 compute nodes from the pool of 9. Our first option is to specify the first available allocation scheme (Figure 3.7)

```
launch_allocation = firstavailable
```

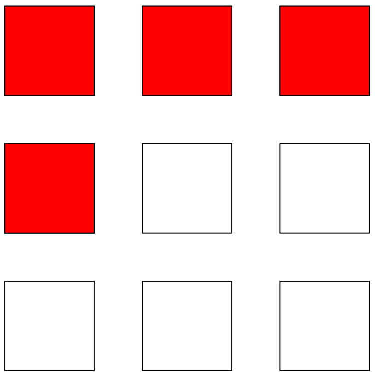


Figure 3.7: First available Allocation of 4 Compute Codes on a 3x3 2D Torus

In first available, the allocator simply loops through the list of available nodes as they are numbered by the topology object. In the case of a 2D torus, the topology numbers by looping through columns in a row. In general, first available will give a contiguous allocation, but it won't necessarily be ideally structured.

To give more structure to the allocation, a Cartesian allocator can be used (Figure 3.8).

```
launch_allocation =   cartesian
cart_launch_sizes = 2 2
cart_launch_offsets = 0 0
```

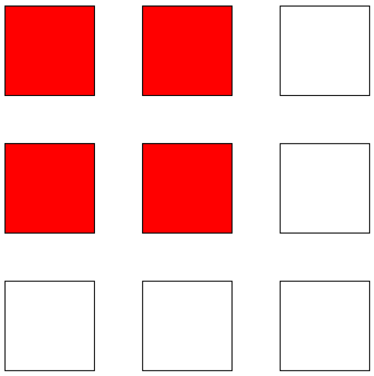


Figure 3.8: Cartesian Allocation of 4 Compute Codes on a 3x3 2D Torus

Rather than just looping through the list of available nodes, we explicitly allocate a 2x2 block from the torus. If testing how “topology agnostic” your application is, you can also choose a random allocation.

```
launch_allocation = random
```

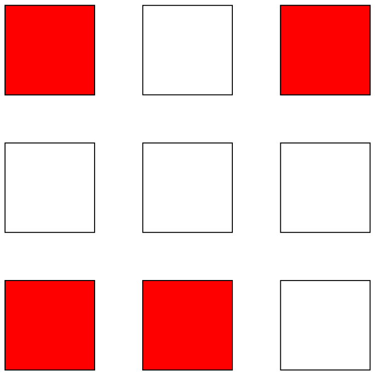


Figure 3.9: Random Allocation of 4 Compute Codes on a 3x3 2D Torus

In many use cases, the number of allocated nodes equals the total number of nodes in the machine. In this case, all allocation strategies allocate the same *set* of nodes, i.e. the whole machine. However, results may still differ slightly since the allocation strategies still assign an initial numbering of the node, which means a random allocation will give different results from Cartesian and first available.

Indexing Schemes

Once nodes are allocated, the MPI ranks (or equivalent) must be assigned to physical nodes, i.e. indexed. The simplest strategies are block and round-robin. If only running one MPI rank per node, the two strategies are equivalent, indexing MPI ranks in the order received from the allocation list. If running multiple MPI ranks per node, block indexing tries to keep consecutive MPI ranks on the same node (Figure 3.10).

```
launch_indexing = block
```

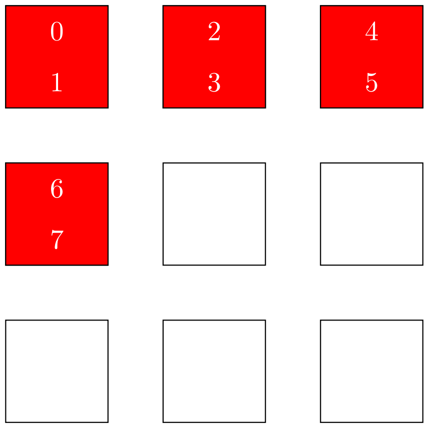


Figure 3.10: Block Indexing of 8 MPI Ranks on 4 Compute Nodes

In contrast, round-robin spreads out MPI ranks by assigning consecutive MPI ranks on different nodes (Figure 3.11).

```
launch_indexing = round robin
```

Finally, one may also choose

```
launch_indexing = random
```

Random allocation with random indexing is somewhat redundant. Random allocation with block indexing is *not* similar to Cartesian allocation with random indexing. Random indexing on a Cartesian allocation still gives a contiguous block of nodes, even if consecutive MPI ranks are scattered around. A random allocation (unless allocating the whole machine) will not give a contiguous set of nodes.

3.9 Using DUMPI

3.9.1 Building DUMPI

As noted in the introduction, SST/macro is primarily intended to be an on-line simulator. Real application code runs, but SST/macro intercepts calls to communication (MPI) and computation functions to simulate

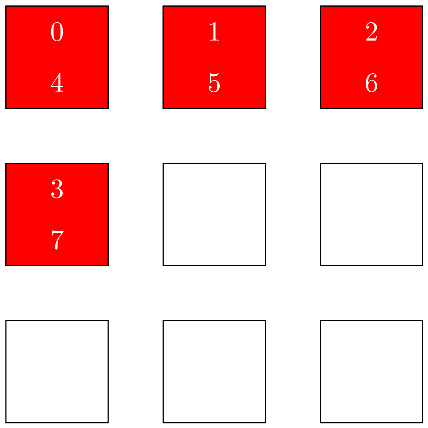


Figure 3.11: Round-Robin Indexing of 8 MPI Ranks on 4 Compute Nodes

time passing. However, SST/macro can also run off-line, replaying application traces collected from real production runs. This trace collection and trace replay library is called DUMPI.

Although DUMPI is automatically included as a subproject in the SST/macro download, trace collection can be easier if DUMPI is built independently from SST/macro. The code can be downloaded from `https://bitbucket.org/jpkenny/dumpi`. If downloaded through Mercurial, one must initialize the build system and create the configure script.

```
dumpi $ ./bootstraps.h
```

DUMPI must be built with an MPI compiler.

```
dumpi/build $ ../configure CC=mpicc CXX=mpicxx \
              --enable-libdumpi --prefix=$DUMPI_PATH
```

The `--enable-libdumpi` flag is needed to configure the trace collection library. After compiling and installing, a `libdumpi.$prefix` will be added to `$DUMPI_PATH/lib`.

Collecting application traces requires only a trivial modification to the standard MPI build. Using the same compiler, simply add the DUMPI library path and library name to your project's `LDFLAGS`.

```
your_project/build $ ../configure CC=mpicc CXX=mpicxx \
                              LDFLAGS="-L$DUMPI_PATH/lib -ldumpi"
```

Trace Collection

DUMPI works by overriding *weak symbols* in the MPI library. In all MPI libraries, functions such as `MPI_Send` are only weak symbol wrappers to the actual function `PMPI_Send`. DUMPI overrides the weak symbols by implementing functions with the symbol `MPI_Send`. If a linker encounters a weak symbol and regular symbol with the same name, it ignores the weak symbol. DUMPI functions look like

```
int MPI_Send(...)
{
  /** Start profiling work */
  ...
  int rc = PMPI_Send(...);
  /** Finish profiling work */
  ...
  return rc;
}
```

collecting profile information and then directly calling the PMPI functions.

We examine DUMPI using a very basic example program.

```
#include <mpi.h>
int main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);
    MPI_Finalize();
    return 0;
}
```

After compiling the program named `test` with DUMPI, we run MPI in the standard way.

```
example$ mpiexec -n 2 ./test
```

After running, there are now three new files in the directory.

```
example$ ls dumpi*
dumpi-2013.09.26.10.55.53-0000.bin
dumpi-2013.09.26.10.55.53-0001.bin
dumpi-2013.09.26.10.55.53.meta
```

DUMPI automatically assigns a unique name to the files from a timestamp. The first two files are the DUMPI binary files storing separate traces for MPI rank 0 and rank 1. The contents of the binary files can be displayed in human-readable form by running the `dumpi2ascii` program, which should have been installed in `$DUMPI_PATH/bin`.

```
example$ dumpi2ascii dumpi-2013.09.26.10.55.53-0000.bin
```

This produces the output

```
MPI_Init entering at walltime 8153.0493, cputime 0.0044 seconds in thread 0.
MPI_Init returning at walltime 8153.0493, cputime 0.0044 seconds in thread 0.
MPI_Finalize entering at walltime 8153.0493, cputime 0.0045 seconds in thread 0.
MPI_Finalize returning at walltime 8153.0498, cputime 0.0049 seconds in thread 0.
```

The third file is just a small metadata file DUMPI uses to configure trace replay.

```
hostname=deepthought.magrathea.gov
numprocs=2
username=slartibartfast
starttime=1380218153
fileprefix=dumpi-2013.09.26.10.55.53
version=1
subversion=1
subsubversion=0
```

Trace Replay

To replay a trace in the simulator, a small modification is required to the example input file in 3.2. We have two choices for the trace replay. First, we can attempt to *exactly* replay the trace as it ran on the host machine.

In this case, we must set the indexing and allocation parameters to use DUMPI. Additionally, we specify the special trace replay application `parsedumpi`. Since we ran with two MPI ranks, we have to set `launch_app1_size = 2`. To configure the trace replay, DUMPI must be given the name of the metafile. So the parameter file might look something like this:

```
# Launch parameters
launch_name = instant
launch_indexing = dumpi
launch_allocation = dumpi
launch_app1 = parsedumpi
launch_app1_size = 2
launch_dumpi_mapname = deepthought.map
launch_dumpi_metaname = dumpi-2013.09.26.10.55.53.meta
# Machine parameters
topology_name = torus
topology_geometry = 2 2
```

After specifying launch parameters, we must specify the machine. In this case, we assume a 2D torus with four nodes. DUMPI records the hostname of each MPI rank during trace collection. In order to replay the trace, the mapping of hostname to coordinates must be given in a node map file, specified by the parameter `launch_dumpi_mapname`. The node map file has the format

```
4 2
nid0 0 0
nid1 0 1
nid2 1 0
nid3 1 1
```

where the first line gives the number of nodes and number of coordinates, respectively. Each hostname and its topology coordinates must then be specified. More details on building hostname maps are given below.

We can also use the trace to experiment with new topologies to see performance changes. Suppose we want to test a crossbar topology.

```
# Launch parameters
launch_name = instant
launch_indexing = block
launch_allocation = firstavailable
launch_app1 = parsedumpi
launch_app1_size = 2
launch_dumpi_metaname = dumpi-2013.09.26.10.55.53.meta
# Machine parameters
topology_name = crossbar
topology_geometry = 4
```

We no longer use the DUMPI allocation and indexing. We also no longer require a hostname map.

Building the Hostname Map

Not all HPC machines support topology queries. The current scheme is only valid for Cray machines, which support topology queries via `xtdb2proc`. SST/macro comes with a script in the bin folder, `xt2nodemap.pl`, that parses the Cray file into the DUMPI format. We first run

```
xtdb2proc -f - > db.txt
```

to generate a Cray-formatted file `db.txt`. Next we run the conversion script

```
xt2nodemap.pl -t hdtorus < db.txt > nodemap.txt
```

generating the hostname map.

3.10 Call Graph Visualization

Generating call graphs requires a special build of SST/macro.

```
build$ ../configure --prefix=$INSTALL_PATH --enable-graphviz
```

The `--enable-graphviz` flag defines an instrumentation macro throughout the SST/macro code. This instrumentation must be *compiled* into SST/macro. In the default build, the instrumentation is not added since the instrumentation has a high overhead. However, SST/macro only instruments a select group of the most important functions so the overhead should only be 10-50%. After installing the instrumented version of SST/macro, a call graph is collected by running

```
sstmac -f parameters.ini -d "<stats> graphviz"
```

with the extra `-d <stats>` flag. After running, a `callgrind.out` file should appear in the folder.

To visualize the call graph, you must download KCachegrind: `http://kcachegrind.sourceforge.net/html/Download.html`. KCachegrind is built on the KDE environment, which is simple to build for Linux but can be very tedious for Mac. The download also includes a QCachegrind subfolder, providing the same functionality built on top of Qt. This is highly recommended for Mac users.

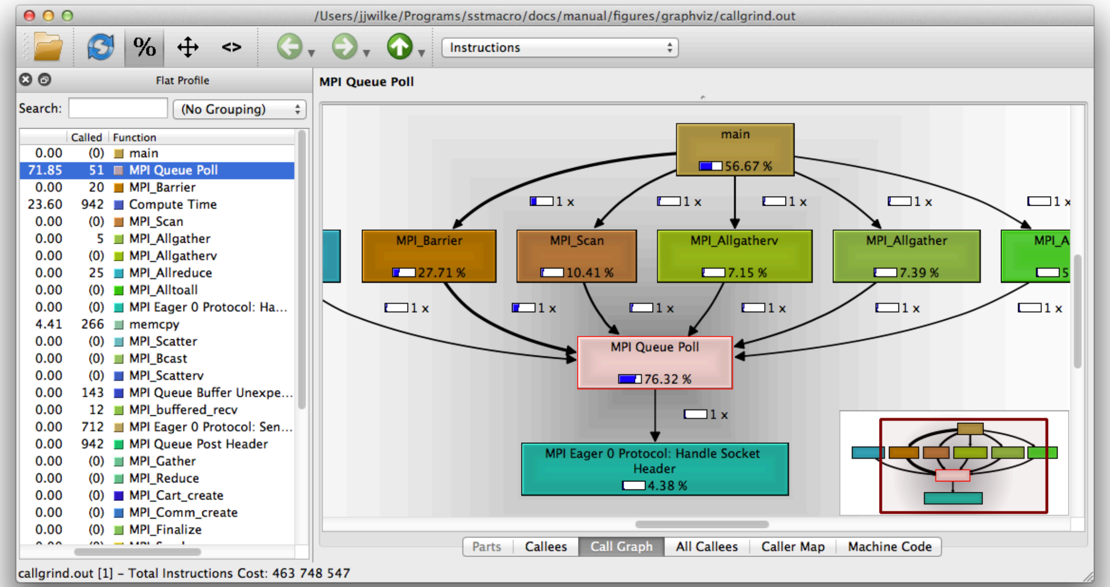
The basic QCachegrind GUI is shown in Figure 3.12. On the left, a sidebar contains the list of all functions instrumented with the percent of total execution time spent in the function. In the center pane, the call graph is shown. To navigate the call graph, a small window in the bottom right corner can be used to change the view pane. Zooming into one region (Figure 3.13), we see a set of MPI functions (Barrier, Scan, Allgatherv). Each of the functions enters a polling loop, which dominates the total execution time. A small portion of the polling loop calls the "Handle Socket Header" function. Double-clicking this node unrolls more details in the call graph (Figure 3.14). Here we see the function splits execution time between buffering messages (memcpy) and posting headers (Compute Time).

3.11 Spyplot Diagrams

Spyplots visualize communication matrices, showing either the number of messages or number of bytes sent between two network endpoints. They are essentially contour diagrams, where instead of a continuous function $F(x, y)$ we are plotting the communication matrix $M(i, j)$. An example spyplot is shown for a simple application that only executes an MPI_Allreduce (Figure 3.15). Larger amounts of data (red) are sent to nearest neighbors while decreasing amounts (blue) are sent to MPI ranks further away.

The spyplot is activated by a "<stats>" flag.

```
sstmac -f parameters.ini -d ``<stats> spyplot"
```



/Users/jjwilke/Programs/sstmacro/docs/manual/figures/graphviz/callgrind.out

Flat Profile

Search: (No Grouping)

	Called	Function
0.00	(0)	main
71.85	51	MPI Queue Poll
0.00	20	MPI_Barrier
23.60	942	Compute Time
0.00	(0)	MPI_Scan
0.00	5	MPI_Allgather
0.00	(0)	MPI_Allgatherv
0.00	25	MPI_Allreduce
0.00	(0)	MPI_Alltoall
0.00	(0)	MPI Eager 0 Protocol: Ha...
4.41	266	memcpy
0.00	(0)	MPI_Scatter
0.00	(0)	MPI_Bcast
0.00	(0)	MPI_Scatterv
0.00	143	MPI Queue Buffer Unexpe...
0.00	12	MPI_buffered_recv
0.00	712	MPI Eager 0 Protocol: Sen...
0.00	942	MPI Queue Post Header
0.00	(0)	MPI_Gather
0.00	(0)	MPI_Reduce
0.00	(0)	MPI_Cart_create
0.00	(0)	MPI_Comm_create
0.00	(0)	MPI_Finalize

callgrind.out [1] – Total Instructions Cost: 463 748 547

Figure 3.12: QCachegrind GUI

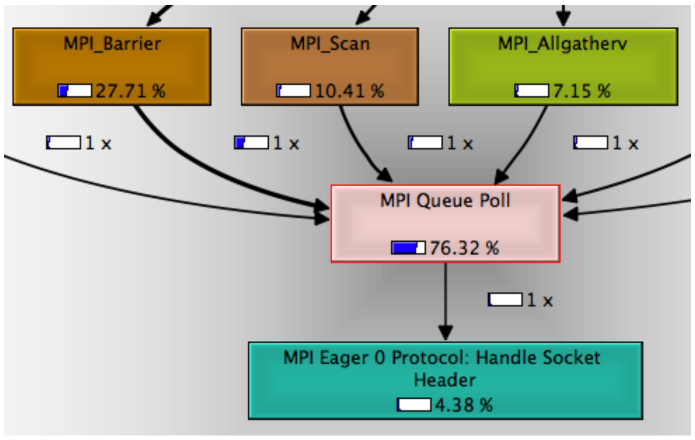


Figure 3.13: QCachegrind Call Graph of MPI Functions

After running there will be three .png files in the folder

```
example$ ls *.png
mpi_spyplot.png
allocation_spyplot.png
machine_spyplot.png
```

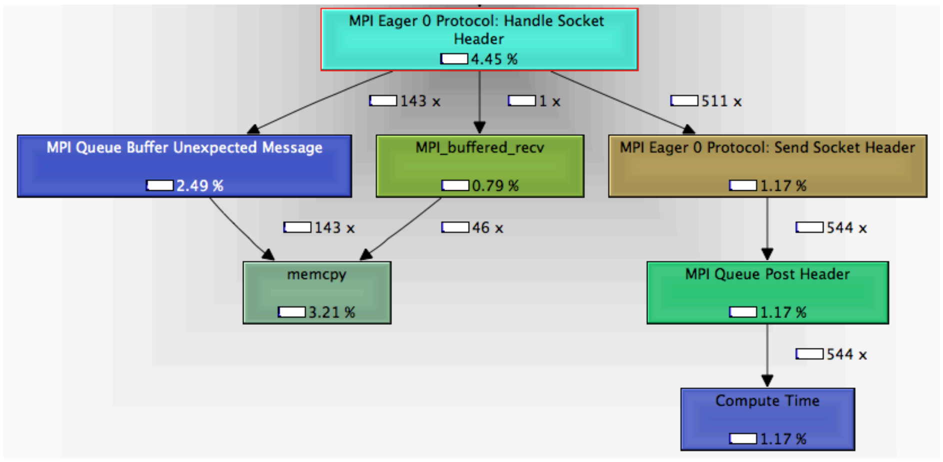


Figure 3.14: QCachegrind Expanded Call Graph of Eager 0 Function

`mpi_spyplot.png` shows the number of bytes exchanged between MPI ranks. `allocation_spyplot.png` shows the number of bytes exchanged between physical nodes, accumulating together all MPI ranks sharing the same node. This gives a better sense of spatial locality when many MPI ranks are on the same node. The plot only includes nodes actually used by application (i.e. included in the allocation). `machine_spyplot.png` is equivalent to `allocation_spyplot.png`, but includes all nodes in the machine even if not included in the allocation.

3.12 Fixed-Time Quanta Charts

Another way of visualizing application activity is a fixed-time quanta (FTQ) chart. While the call graph gives a very detailed profile of what code regions are most important for the application, they lack temporal information. The FTQ histogram gives a time-dependent profile of what the application is doing (Figure 3.16). This can be useful for observing the ratio of communication to computation. It can also give a sense of how "steady" the application is, i.e. if the application oscillates between heavy computation and heavy communication or if it keeps a constant ratio. In the simple example, Figure 3.16, we show the FTQ profile of a simple MPI test suite with random computation mixed in. In general, communication (MPI) dominates. However, there are a few compute-intensive and memory-intensive regions.

The FTQ visualization is activated by a "<stats>" flag.

```
sstmac -f parameters.ini -d ``<stats> ftq''
```

After running, two new files appear in the folder: `plot_app1.p` and `ftq_app1.dat`. `plot_app1.p` is a Gnuplot script that generates the histogram as a postscript file.

```
gnuplot plot_app1.p > output.ps
```

Gnuplot can be downloaded from `http://www.gnuplot.info` or installed via MacPorts. We recommend version 4.4, but at least 4.2 should be compatible.

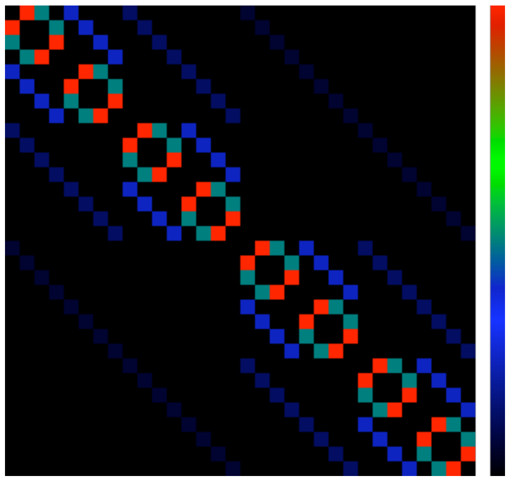


Figure 3.15: Spyplot of Bytes Transferred Between MPI Ranks for MPI_Allreduce

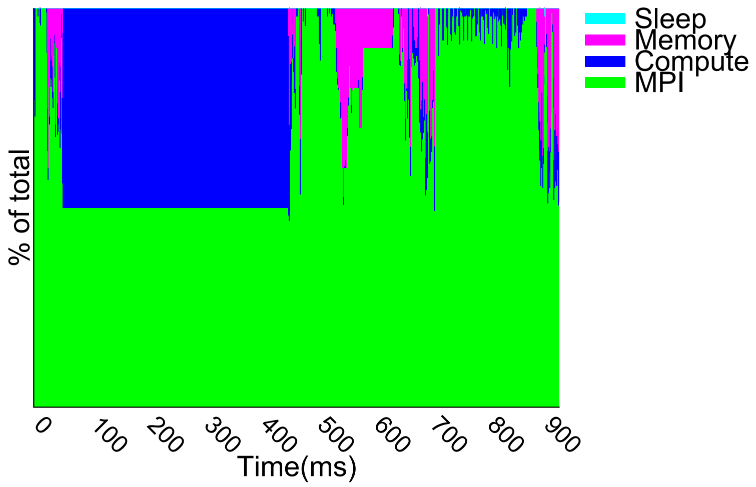


Figure 3.16: Application Activity (Fixed-Time Quanta; FTQ) for Simple MPI Test Suite

The granularity of the chart is controlled by the `ftq_epoch` parameter in the input file. The above figure was collected with

```
ftq_epoch=5us
```

Events are accumulated into a single data point per "epoch." If the timestamp is too small, too little data will be collected and the time interval won't be large enough to give a meaningful picture. If the timestamp is too large, too many events will be grouped togther into a single data point, losing temporal structure.

Chapter 4

Applications and Skeletonization

4.1 Basic Application porting

To conduct on-line simulation of an user application, one needs to compile and link the application's source code with the SST/macro library. A few modifications to the source code are needed for correct compilation and execution.

The first requirement is to change the application's entry point. The SST/macro framework has already taken the `main` function, and consequently the user application becomes a sub-routine within the simulation environment. As introduced in Section 3.3, one needs to change the entry function of the user application from `main` to `user_skeleton_main`, which has the same function signature as the `main` function.

The second requirement is to replace the inclusions of the MPI header file with its SST/macro counterpart. This can be done by either replacing all occurrences of `#include <mpi.h>` with `#include <sstmac/sstmpi.h>`, or have a wrapper header file named `mpi.h` that does the inclusion of `sstmac/sstmpi.h` in the local source directory. A more general `#include <sstmacro.h>` can also be used, which pulls in `mpi.h` and some other useful things.

The last requirement is optional, which may apply to C applications that use file scope or program scope global variables. The necessary modifications are detailed in the following section.

4.1.1 Global variables

SST/macro simulates multi-node MPI programs on a single node by executing each MPI process in a separate thread. Thread execution, though sharing many similarities with process execution, lacks private memory space. When porting an application to SST/macro, global variables used in C programs will not be mapped to separate memory addresses causing incorrect execution or even segmentation faults. Global variables need to be either removed through refactoring or replaced by a thread-safe version. SST/macro provides a complete set of global variable replacements from `#include <sstmac/software/process/sstmac_global.h>`. Include this header file prior to the declaration of your global variables and replace the variable type declaration with the ones that have a `global_` prefix in the header file. By including this file, you now must compile your application with a C++ compiler as a C++ program. While most of C++ is backwards-compatible, there are some things that are not, and will require either a compiler flag to relax strictness or quick refactor of some of your syntax.

When printing a global variable with `printf`, the user should explicitly invoke a cast to the primitive type in the function call:

```
print("Hello world on rank %d", int(rank));
```

If not explicitly cast, the `va_args` function will be misinterpreted and produce an "Illegal instruction" error. This still follows the "single-source" principle since whether compiling for SST/macro or a real machine, the code is still valid.

4.2 Manual Skeletonization

A program skeleton is a simplified program derived from a parent application. The purpose of a skeleton application is to retain the performance characteristics of interest. At the same time, program logic that is orthogonal to performance properties is removed. The rest of this chapter will talk about skeletonizing an MPI program, but the concepts mostly apply regardless of what programming/communication model you're using.

The default method for skeletonizing an application is *manually*. In other words, going through your application and removing all the computation that is not necessary to produce the same communication/parallel characteristics. Essentially, what you're doing is visually backtracing variables in MPI calls to where they are created, and removing everything else. This is what the auto-skeletonizer does using compiler static analysis, described in Section 4.3.

Skeletonization falls into three main categories:

- *Data structures* - Memory is a precious commodity when running large simulations, so get rid of every memory allocation you can.
- *Loops* - Usually the main brunt of CPU time, so get rid of any loops that don't contain MPI calls or calculate variables needed in MPI calls.
- *Communication buffers* - While you can pass in real buffers with data to SST/macro MPI calls and they will work like normal, it is relatively expensive. If they're not needed, get rid of them.

A decent example of skeletonization is HPCCG_full (the original code) and HPCCG_skel (the skeleton) in sstmacro/skeletons.

4.2.1 Compute Modeling

By default, even if you don't remove any computation, *simulation time doesn't pass between MPI or other calls implemented by SST/macro* unless you set

```
host_compute_modeling = true
```

in your parameter file. In this case, SST/macro will use the wall time that the host takes to run code between MPI calls and use that as simulated time. This only makes sense, of course, if you didn't do any skeletonization and the original code is all there.

If you do skeletonize your application and remove computation, you need to replace it with a model of the time or resources necessary to perform that computation so that SST/macro can advance simulation time properly. There are 3 ways of doing this, outlined in the following sections.

Absolute time model

You can describe the time it takes to do computation by inserting calls to

```
void SSTMAC_compute(double seconds)
```

Usually, this would be parameterized by some value coming from the application, like loop size. You can also describe memory movement with

```
void SSTMAC_block_read(long long bytes)
```

again usually parameterized by something like vector size. Using these two functions is the simplest and least flexible way of compute modeling.

Linear loop model

For convenience, we also have

```
void SSTMAC_compute_loop(long lower, long upper, int fudge)
```

which models a loop by coarsely simulating both instructions and memory movement. The *lower* and *upper* parameters describe loop bounds, and *fudge* is an estimate of how much work goes on inside the loop. We usually estimate *fudge* by simple counting the lines of code in the loop, and modifying it from there based on validation results. To use this compute model, you should put the following in your parameter file:

```
compute_loops_enable = true
compute_loops_mem_ratio = 0.3
compute_loops_flop_ratio = 0.3
```

where the mem and flop ratio parameters are numbers between 0 and 1 that control how memory or compute-intensive the calls are (they are globally applied to all compute loops calls). Generally, the formula for memory bytes transferred (B) and compute instructions performed (C) is:

$$B = (upper - lower) * fudge * memratio$$
$$C = (upper - lower) * fudge * flopratio$$

There are also convenience functions for nested loops, like

```
void SSTMAC_compute_loops2(int low1, int up1, int low2, int up2, int fudge)
```

and so on.

Eiger

The most flexible, but most complex, method for modeling computation is to use the Eiger Statistical Modeling framework to generate performance models that are polled at runtime. These models take as parameters application-level metrics and map them to performance based on statistical models generated from instrumentation data collected off-line from simulation with SST/macro. These models are generated based off empirical data; patterns about the effect variations in the input values have on performance appear as the sample size grows. This means that by collecting instrumentation data for a large range of application problem sizes, the models are capable of extrapolating performance as problem size grows. Typically the input metrics for these models are any values that may have a direct or indirect effect on the performance of the application. For example, it would be safe to assume that the matrix sizes for a matrix multiplication kernel would correlate with execution time, and therefore would be a good candidate for inclusion in instrumentation. The Eiger framework is robust to handling large sets of input metrics, so the rule of thumb is to be liberal when deciding what to include.

The method for polling Eiger models is through

```
void SSTMAC_compute_eiger(std::map<std::string, double> values, std::string model_name)
```

The *values* parameter maps metric names to their associated values to poll the model *model_name*. We also provide convenience functions

```
void SSTMAC_compute_eiger1(std::string model_name, std::string prefix,
                           std::string name1, double val1);
void SSTMAC_compute_eiger2(std::string model_name, std::string prefix,
                           std::string name1, double val1, std::string name2, double val2);
void SSTMAC_compute_eiger3(std::string model_name, std::string prefix,
                           std::string name1, double val1, std::string name2, double val2,
                           std::string name3, double val3);
```

and so on up to 7. For these versions, the *prefix* parameter allows annotating a prefix to the file name containing the model.

Example usage of Eiger in a skeleton is minimd-cpu, found in sstmacro/skeletons.

Please visit `https://bitbucket.org/eanger/eiger` for more information on getting, installing, and using Eiger to generate models, as well as methodology for adding instrumentation to your code.

4.2.2 Skeletonization Issues

The main issue that arises during skeletonization is data-dependent communication. In many cases, it will seem like you can't remove computation or memory allocation because MPI calls depend somehow on that data. The following are some examples of how we deal with those:

- *Loop convergence* - In some algorithms, the number of times you iterate through the main loop depends on an error converging to near zero, or some other converging mechanism. This basically means you can't take out anything at all, because the final result of the computation dictates the number of loops. In this case, we usually set the number of main loop iterations to a fixed (parameterized) number. Do we really care exactly how many loops we went through? Most of the time, no, it's enough just to produce the behavior of the application.

- *Particle migration* - Some codes have a particle-in-cell structure, where the spatial domain is decomposed among processes, and particles or elements are distributed among them, and forces between particles are calculated. When a particle moves to another domain/process (because it's moving through space), this usually requires communication that is different from the force calculation, and thus depends entirely on the data in the application. We can handle this in two ways:
  1. *Ignore it* - If it doesn't happen that often, maybe it's not significant anyway. So just remove the communication, recognizing that the behavior of the skeleton will not be fully reproduced.
  2. *Approximate it* - If all we need to know is that this migration/communication happens sometimes, then we can just make it happen every so many iterations, or even sample from a probability distribution.
- *AMR* - Some applications, like adaptive mesh refinement (AMR), exhibit communication that is entirely dependent on the computation. In this case, skeletonization is basically impossible, so you're left with the following options:
  - *Traces* - revert to DUMPI traces, where you will be limited by existing machine size. Trace extrapolation is also an option here.
  - *Run it* - get yourself a few servers with a lot of memory, and run the whole code in SST/macro.
  - *Synthetic* - It may be possible to replace communication with randomly-generated data and decisions, which emulate how the original application worked. This hasn't been tried yet.
  - *Hybrid* - It is possible to construct meta-traces that describe the problem from a real run, and read them into SST/macro to reconstruct the communication that happens. Future versions of this manual will have more detailed descriptions as we formalize this process.

4.3 Semi-Automatic Skeletonization

The skeletonization process described here is an iterative process to generate a program skeleton from source code. The skeleton extraction process consists of several sub-processes:

1. Static code analysis.
2. User guidance to augment analysis.
3. API Specification of skeletonization target characteristics.
4. Code transformation and generation.

4.3.1 Building the Auto-skeletonizer

The latest version of the auto-skeletonizer is shipped as part of the ROSE compiler project. It will be build automatically when building the ROSE compiler from source, which can be obtained from GitHub:

```
$ git co https://github.com/rose-compiler/edg4x-rose.git
```

For detailed instructions on how to build the ROSE compiler, please refer to the ROSE documentation on: `http://rosecompiler.org/`. If an old version of ROSE installation is avaible to your enviroment, the auto-skeletonizer might also be build separeately using the existing ROSE installation. First, obtain the source code of the stand alone auto-skeletonizer using git:

```
$ git co https://github.com/mjsottile/rose-mpi-skeletons.git rose-mpi-skeletons
```

Then modify the makefile inside the `rose-mpi-skeletons` folder making it pointing to the existing ROSE installation. Edit the first two lines of the makefile as follows:

```
ROSEINSTALL=/path/to/your/rose/installation/dir/
BOOSTINTALL=/path/to/your/boost/installation/dir/
```

Last, building the executables with the default makefile:

```
$ make
```

A successful build using either approach should give your three executables: extractMPISkeleton, generateSignatures, and summarizeSignatures. The use of these executables to generate skeleton source code is explained in the following section.

4.3.2 Static Analysis

In the static analysis phase, the compilation analysis is performed in three parts:

1.1 Generating summary information for each function in a given compilation unit.

1.2 Combining individual summary files into a larger summary of the program.

1.3 Extraction of the MPI skeleton using the combined summary information.

In 1.1, in order to generate the summary information of a source program, we can invoke the following tool:

```
 generateSignatures [options] file ...
```

The command-line options are as follows:

```
 -signature:(o|output) filename - Use 'filename' as the output.
 -signature:(d|debug)           - Print debugging messages.
```

In 1.2, we can combine several compilation units from 1.1 to generate a combined signature by invoking the following tool:

```
 summarizeSignatures [options] <signature files>
```

The command-line options are as follows:

```
 -summarize:(o|output) filename - Use 'filename' as the output.
 -summarize:(s|spec) filename   - Use 'filename' as API specification.
 -summarize:(d|debug)           - Print debugging messages.
```

The last two options are only needed for debugging.

The user should specify all the signature files from the previous step of the analysis in a single step. While the generation of the summary is additive, the overall analysis is more efficient if performed in a single step.

In 1.3, the user can extract the program skeleton using the summarized signature file from 1.2 by invoking the following tool:

```
extractMPISkeleton [options] file ...
```

The filenames are skeletonized and the output is written to files with 'rose ' prepended to the filenames.

The command line options are these:

```
 -skel:(o|outline)       - Outline everything not in the skeleton.
 -skel:(s|spec) filename - Use 'filename' as the specification of the API.
 -skel:(g|sig)  filename - Use 'filename' as the summary information.
 -skel:(d|debug)         - Print debugging messages.
 -skel:(p|pdf)           - Generate PDF of the generated AST.
```

The last two options are only needed for debugging. In order for the analysis to be complete, a signature file generated from the summarizeSignatures tool must be provided.

Current limitations: The skeleton generator supports C and C++ only. Fortran is not supported with this version of the program skeletonization tool.

4.3.3 User guidance to augment analysis

In this phase, users can annotate the program skeleton file to augment the code analysis.

Annotations used by this tool are specified in the following format:

```
#pragma skel [specific pragma text here]
```

Currently, three types of data annotation are supported:

Loop annotations

It is not uncommon for skeletonized code to no longer have the looping behavior of the parent application.

To fix any problems stemming from the modified loop behavior, three options for loop annotations are available:

```
#pragma skel loop iterate exactly(n)
#pragma skel loop iterate atmost(n)
#pragma skel loop iterate atleast(n)
```

These options correspond to forcing an exact, upper, and lower bound on the iteration count. The pragma must be placed immediately preceding the loop of interest. Loops constructed with 'for', 'while', or 'do while' are all supported as well as loops containing break and continue statements.

Data declaration annotations

If a program contains an array that should be preserved in the skeleton, it is useful to have control over how it is initialized since often the skeleton will not contain the computation code that populates the array elements. The initializer pragma allows these element values to be specified.

```
#pragma skel initializer repeat(x)
int myArray[14];
```

(Where ’x’ is a C-expression interpreted in the current scope of the program.)

Conditional statement annotations

The conditional annotation currently allows programmers to experiment with skeletons that will randomly branch one way or the other with a specified probability.

The general case is:

```
#pragma skel condition prob(p)
```

(Where ’p’ is a C-expression interpreted in the current scope of the program which should be a floating point number between 0 and 1.0. As noted above, floating point constants are not allowed due to current limitations of ROSE.)

4.3.4 API Specification of skeletonization points

The skeleton generator skeletonizes programs relative to one or more API specifications.

APIs are specified in a configuration file that uses an s-expression format. Each API call is a sub-expression with the format:

```
(API_FUNCTION_NAME ARGUMENT_COUNT (deptype argA ..) (deptype argB ..) ...)
```

The API collection file is specified using the -skel:s command line option:

```
$ extractMPISkeleton −skel:s /where/is/the/collection/file
```

4.3.5 Outlining

The skeleton generator can work in an alternate mode, Outlining. This mode is specified with the following command-line option:

```
  -skel:o
  -skel:outline
```

In this mode, rather than removing the non-skeleton code, the tool will ”outline” (move into separate functions, the converse of inlining) all non-skeleton code.

4.3.6 Skeleton Behavior Validation

Skeleton correctness is important in studying the scalability of an application using the skeleton-driven approach. Based on the trace files produced by the DUMPI library in Section 3.9, the trace analysis tool creates a statistical report of various metrics and characteristics of the program skeleton.

The trace analysis tool is built as part of the SST/macro package and can be found in the installation directory. The following command will invoke the trace analysis tool on an existing trace file:

```
$ <SSTMAC_INSTALL>/bin/traceanalyzer -v -o report.xml dumpitracer.meta
```

A XML report file will be generated after the analyzing process is done. When comparing the report file with one generated from the original applicaiton, the MPI message and traffic count can be used to verify if the desired communication behavior is preserved.

Chapter 5

Uncertainty Quantification Methods and Tools

5.1 Overview

For uncertainty quantification (UQ) and validation studies the UQ Toolkit (UQTk) library is being used. UQTk (`www.sandia.gov/UQToolkit`) is a lightweight C++ library, developed in Sandia National Laboratories, California, that primarily offers tools for surrogate model construction and uncertainty propagation with polynomial chaos expansions (PCE), as well as model calibration and validation.

UQTk Version 2.0 is released under the GNU Lesser General Public License (LGPL).

5.2 Download and compilation

A tar-ball with the source code, tutorials, examples and documentation can be downloaded from `www.sandia.gov/UQToolkit/uqtk_download.html`

Before compilation, one should set a file `config/config.site` with computer-specific compilation options. Both C++ and Fortran compilers are required. The file `config/config.gnu` provides a good starting point as a sample configuration file.

Compilation is simply invoked by

```
make all
```

5.3 Structure

Below we present parts of the UQTk structure that are relevant to UQ and validation studies on SST/macro.

- `config`: configuration file location
- `doc_cpp`: doxygen documentation
- `src_cpp`: source code - all libraries reside here

  `src_cpp/lib`: location of all compiled libraries in C++ and Fortran

  `src_cpp/apps`: command line utilities for performing various UQ tasks (source codes)

  `src_cpp/bin`: command line utilities for performing various UQ tasks (executables)

- `examples_cpp`: example codes and scripts

  `examples_cpp/uq_surr`: a set of scripts that automates the forward UQ task utilizing the command line `apps` above.

5.4 Applications/Capabilities

The two basic UQ tasks, enabled by UQTk, are *forward UQ* and *inverse UQ* as outlined in the next subsections.

5.4.1 Forward UQ: uncertainty propagation and global sensitivity analysis

The main technique we employ for forward UQ is the spectral Polynomial Chaos expansions (PCEs). A PCE for a given model allows for a) efficient uncertainty propagation, b) very fast global sensitivity analysis, and c) cheap surrogate model construction that can replace the original model in sampling-intensive studies such as calibration, optimization, or, generally, inverse UQ.

The PCE library consists of a class for PC objects allowing both intrusive and non-intrusive uncertainty propagation. For SST/macro, only non-intrusive methods are feasible that deal with the SST/macro model as a *black-box* without the need to rewrite or reformulate it.

Generic workflow:

A set of scripts that illustrate the essential forward UQ tasks is located in `examples_cpp/uq_surr`. To enable running all scripts one should set an environment variable `UQTK_SRC` that points to the location of UQTk.

```
setenv UQTK_SRC location/of/uqtk/in/your/computer
```

A generic workflow consists of 4 steps:

1. Generate parameter samples to run the forward model at, for PC construction and validation, `gen_sam.x`

   ```
   gen_sam.x <domain_file> <sampling_type(Q/U)> <N_samples> <N_val>
   ```

The list of arguments:

domain_file: A file with d rows and 2 columns. d is the total number of parameters being explored. The two columns are the lower and upper bound of the corresponding parameter.

sampling_type: Q (Quadrature) or U (uniformly random). Note that currently in this script set, only Q is supported for further PCE generation.

N_samples: Number of samples used for training, i.e. for building PCE.

N_val: Number of random samples generated for PCE validation. This can be set to 0 to skip validation.

2. Run the black-box model, `model.x`

```
model.x
```

This is a black-box model that takes no arguments. However, it expects two input files mparam.dat and minput.dat, and returns the function evaluations in the output file moutput.dat.

mparam.dat: a single column $d \times 1$ of the parameters of interest, where d is the number of parameters

minput.dat: controllable input in a matrix form, $N \times k$, where k is the number of controllable parameters and N is the number of values or model observables.

moutput.dat: the model output in a column format $N \times 1$.

This is the file that needs to be modified/provided by the user according to the model under study. Currently, a simple function $y = Ae^{Bx} + 2Bx$ is implemented, where A and B are the parameters (mparam.dat) and x is the single controllable input parameter (minput.dat). A user-created `model.x` should accept input files minput.dat and mparam.dat as described above, and it should produce an output file moutput.dat with the formats described above,

3. Obtain PCE for the model, `uq_surr.x`

```
uq_surr.x <domain_file> <sampling_type(Q/U)> <N_samples> <N_val> \
    <P_order> <moutput_surr_filename> <moutput_val_filename>
```

The first four arguments coincide with those from `gen_sam.x`, the rest of the arguments are:

P_order: the PCE surrogate order. Polynomial series is truncated according to the total order.

moutput_surr_filename: model output file resulting from running `model.x` on training samples.

moutput_val_filename: model output file resulting from running `model.x` on validation samples.

4. Postprocess, e.g. global sensitivity analysis, `pp_sens.x`

```
pp_sens.x
```

The current implementation of the post processing script `pp_sens.x` takes no arguments. It expects the presence of a PCE information file pccf_all.dat, which is one of the outcomes of `uq_surr.x`. The final global sensitivity results are saved in the `allsens.dat` file of dimensions $N \times d$, where each row corresponds to a single value for the controllable input (number of controllable inputs = N), and each column corresponds to the sensitivity index of a parameter (number of parameters = d).

Note that one can run the model `model.x` in an *online* regime by using the keyword "M" instead of filenames in

```
uq_surr.x <domain_file> <sampling_type(Q/U)> <N_samples> <N_val> <P_order> M M
```

that effectively incorporates the steps 1-3 above.

Simple Example:

The script `example.x` incorporates the full workflow above for a test function $y = Ae^{Bx} + 2Bx$. Try

```
example.x online
```

which produces, in a newly created folder `test`, the sensitivity file `allsens.dat` with sensitivity indices with respect to A and B for all values of controllable input x.

How Forward UQ applies to SST/macro

In the workflow described above, SST/macro would replace the black-box simple model given by the equation, and SST/macro parameters of interest would replace A and B. Note that while we could explore the entire SST/macro parameter space using this workflow, the higher the dimensionality of the space the more time process takes, so we would also want to narrow it down to the most important parameters, usually by reasoning, inspection, or simple experimentation. Also note that running an SST/macro simulation can be considerably more expensive than evaluating an analytical model, which is why we typically construct a *surrogate*, or an arbitrarily-complex polynomial that can approximate performance given by SST/macro.

While the above workflow must currently be manually applied to SST/macro data, we are working on automating this process and coming up with tutorials explaining how it is done.

5.4.2 Inverse UQ: parameter calibration and model validation

UQTk relies on Bayesian inference methods for model parameter calibration. Model validation is a direct result of calibration postprocessing. Indeed, a model is considered validated if the calibrated model parameters and the associated uncertainties can explain/predict available data well. The library `uqtkmcmc` allows implementation of Bayesian calibration using Markov chain Monte Carlo (MCMC) methods. The MCMC technique essentially searches the parameter space and compares model results with available data. Note that each parameter sample invokes a model evaluation, and often MCMC requires many samples for properly estimating the uncertainties. In such cases, the full model, as a black-box, will be replaced by its surrogate, again as a black-box model, that is constructed by the forward UQ techniques described in Subsection 5.4.1.

Appendices

Appendix A

Debug Flags

A.1 Debug output

The following are descriptions of flags you can use to turn on debug information about what's happening in the simulator. Some are more verbose than others. Multiple levels of output can be defined, and turned on with ([level]) after the flag, such as -d "<debug> mpi(2)". In general, a higher level is more output, and the default level is 1.

To begin with, specifying -d "<debug> all" will print everything. This is sometimes useful, e.g. if you want to know the last thing that happened before the simulation failed.

Even more, specifying just -d "all" will enable all debug, tracing, and statistics. This is never recommended, as you are not likely to have all the parameters for this set correctly, and it really has no useful purpose.

For turning on individual output, the following flags are generally more useful:

A.1.1 Software

- *os* - prints out everything the operating system is doing, including swapping threads and such (warning: verbose)
- *sstmac_mpi* - print out calls to the MPI C interface
- *library* - prints out anything a library does (lots of things are libraries, so probably very verbose)
- *mpi* - prints out everything mpi does behind the scenes (warning: verbose). Includes the following, which can be turned on individually:
  - *mpicollective* - prints out just what the collectives are doing
  - *mpicommfactory* - prints the activity from creating mpi communicators
  - *mpiqueue* - basic operations in mpi transactions, like src/tag matching
  - *mpiprotocol* - what's going on with the protocols mpi uses (RDMA exchange, etc)

  - *mpiapi* - the underlying functions that implement mpi behind the C interface, may give more information than sstmac_mpi
  - *mpirequest* - prints some info when a transaction (which has an associated mpi request) completes
  - *mpiserver* - some low level interactions with the node model / OS for basic sending of messages and finding the right mpiqueue

- *shmem* - prints out everything that OpenSHMEM does. Includes the following, which can be turned on individually:
  - *api* - calls to the SHMEM api
  - *server* - low level operations in the SHMEM implementation
  - *symmetric* - prints info about operations on symmetric data
  - *heap* - prints info about heap operations (usually on symmetric data structures)
  - *transfer* - prints out what transfers are doing (remote operations are all "transfers" in our implementation).
- *upc* - prints out everything that UPC does. Includes the following, which can be turned on individually:
  - *api* - calls to the UPC api (including implicit ones that may appear after preprocessing)
  - *server* - low level operations in the UPC implementation
  - *shared* - prints info about operations on shared objects
  - *shared_ptr* - prints info about operations on shared pointer objects
  - *shared_heap* - prints info about shared heap operations (usually on shared or shared pointers)
  - *transfer* - prints out what transfers are doing (remote operations are all "transfers" in our implementation).
- *socket* - prints what the socket API/library is doing
- *lib_compute_loops* - prints stuff about compute loops library usage
- *parsedumpicallbacks* - prints information for each MPI call when parsing a DUMPI trace
- *payload* - prints the payload modeling

A.1.2 Hardware

- *network* - prints out everything the network does (warning: verbose). Includes the following, which can be turned on individually:
  - *interconnect* - setup of the interconnect
  - *networkswitch* - if using a discrete switch-based model, prints out everything the switches do
  - *topology* - prints out the connections between switches
  - *routing* - prints routing decisions
  - *trainswitch* - if using the train network model, prints out what the train switch is doing

- *nic* - prints out network interface activity
- *fastflower* - if using the fastflower model, prints out all modeling activity
- *train* - if using the train model, prints out train modeling activity
- *processor* - prints processor/instruction modeling
- *node* - prints everything that goes through the node (which is pretty much all events on a node)
- *memory* - prints memory modeling

A.1.3 Backend

- *sstmac* - setting up the simulation
- *eventmanager* - prints out the processing and scheduling of every event (warning: the most verbose of all)
- *eventupdates* - periodically prints out the event processing rate
- *parallel* - if running PDES, prints information about partitioning, and synchronization activity

A.1.4 Application

- *app* - prints out anything you put in your application, which must have inherited from sstmac::sw::app, and used SSTMAC_DEBUG
- *launch* - prints out everything about which application processes are launching on which nodes, and how they're numbered. Includes the following:
  - *allocation* - prints out the nodes allocated to apps
  - *indexing* - prints out which task ids (e.g. mpi ranks) go on which allocated nodes
- *mpicheck* - if running an MPI application, prints a banner when rank 0 has passed the barrier in MPI_Finalize(). This is usually recommended instead of using final simulation time as it is a more consistent simulation time, especially if running PDES.

A.2 Tracing

The following flags can be used like -d "<trace> *flag*"

- *dumpi* - produces a DUMPI trace from the simulation
- *hpx* - if using HPX, produces a trace of how threads get mapped to cores
- *events* - a generic trace of events that cause thread blocking/unblocking.

A.3 Statistics

The following statistics can be collected using -d "<stats> *flag*". In general, different collected statistics produce different files at the end of the simulation.

- *events* - a backend histogram of the time into the future that messages are scheduled
- *spyplot* - generate spyplots of different levels of communication (MPI, allocation, and machine).
- *spyplot_png* - just generate the png images for the spyplots
- *spyplot_csv* - only generate the csv version of spyplots
- *ftq* - collect finite-time quanta stats. See Section 3.12.
- *callgraph* - collect graphviz callgraphs. See Section 3.10.
- *hpx* - if using HPX, some statistics on HPX message passing
- *hpx_threads* - if using HPX, some statistics about thread creation, etc.
- *processor* - stats on processor usage
- *congestion* - stats about which links are congested and how much
- *latency* - nic-to-nic latency
- *histograms* - message size histograms at nic level
- *mem_bytes* - memory traffic

Appendix B

Compatibility

B.1 Integration Testing

Continuous integration tests are performed for each code check in to the SST/macro and DUMPI mainline development repositories. The latest results of these tests can be viewed at http://sstmacro.ca.sandia.gov:8085. Current build server configurations and the extent of functionality tests are as follows.

B.1.1 Mac OS X Test System (sstmacro)

Mac OS X 10.8.2/Mountain Lion

- gcc 4.2.1
- openmpi 1.6.3
- gmp 5.0.5
- boost 1.48

B.1.2 Mac OS X Test System (sstmac5)

Mac OS X 10.8.2/Mountain Lion

- gcc 4.2.1
- clang 3.3
- gmp 5.0.5
- boost 1.48

B.1.3 Ubuntu Test System (empedocles)

Ubuntu 11.04/natty

- gcc 4.5.2

B.1.4 SST/macro Functionality Tests

- OS X build
- OS X distribution
- OS X documentation
- OS X external boost 1.48
- OS X clang
- OS X gmp
- Ubuntu distribution
- Ubuntu custom new

B.1.5 Dumpi Functionality Tests

- OS X build
- OS X distribution
- OS X documentation
- OS X open mpi

B.2 Other Known Working Systems

B.2.1 Red Hat Production System (warpcore)

Red Hat Enterprise Linux Workstation release 6.2 (Santiago)

- gcc 4.4.6

B.2.2 Typical Development System

Mac OS X 10.8.4/Mountain Lion

- gcc 4.7.2