SST/macro 6.1: User's Manual

Sandia National Labs
Livermore, CA

November 1, 2016

Contents

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 (3.8) 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 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 5.3. 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, to be detailed in a pending SAND report. Skeletonization and UQ are the two main elements in the "canonical" SST/macro workflow (Figure 1.1).
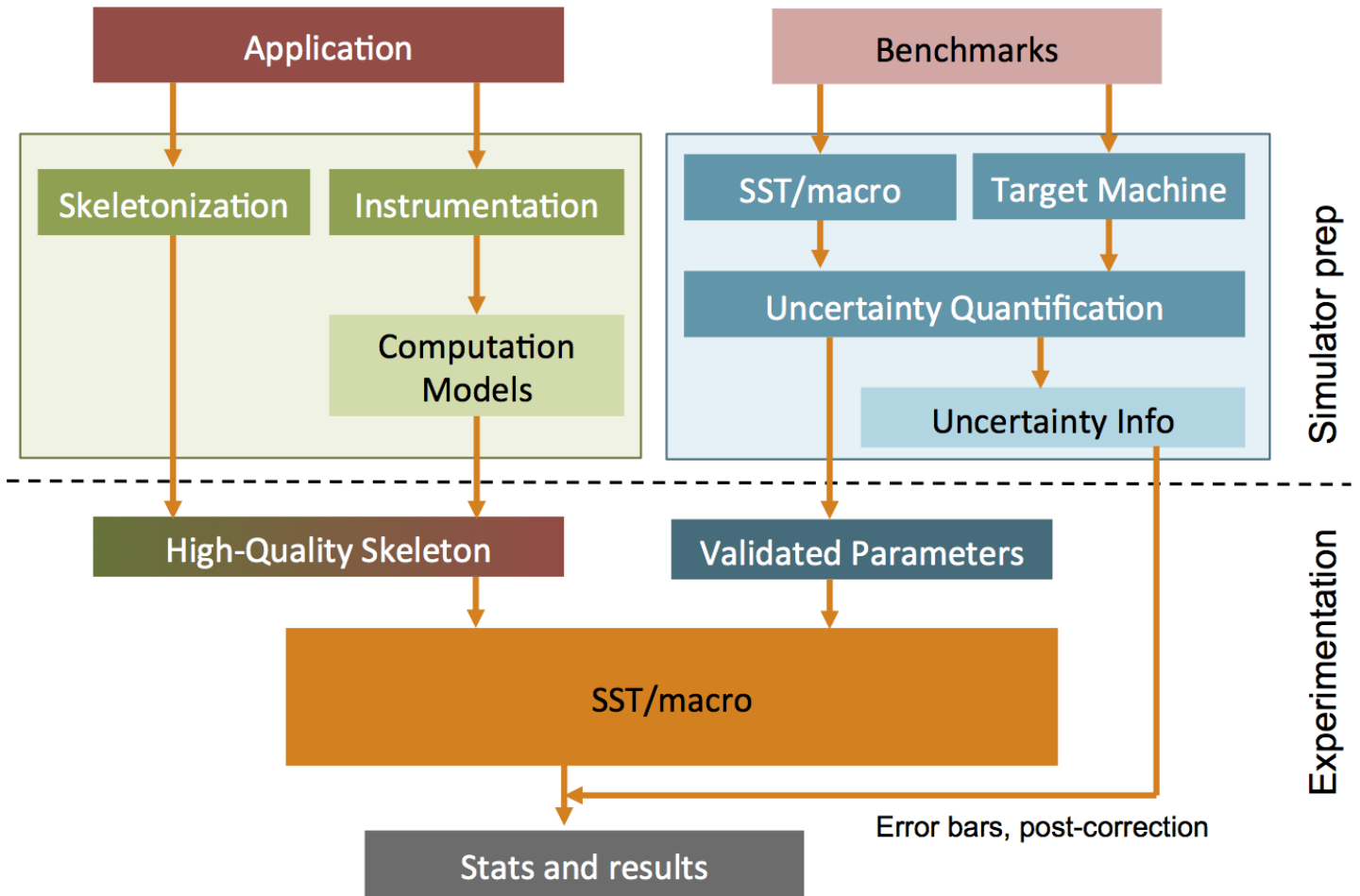
Figure 1.1: SST/macro workflow.

1.2 Currently Supported

1.2.1 Programming APIs

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 `unimplemented_error`, reporting the function name.

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 3.9.

- 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 3.10.

- 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 3.11

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 5.1 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_COMPLEX`) 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:

```
1  while(!flag)
2  {
3    MPI_Iprobe( 0, 0, MPI_COMM_WORLD, &flag, &status );
4  }
```

  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`).

1.3.3 Fortran

SST/macro previously provided some experimental support for Fortran90 applications. This has been discontinued for the foreseeable future. For profiling existing apps written with Fortran, DUMPI traces can still be generated.

Chapter 2

Building and Running SST/macro

2.1 Build and Installation of SST/macro

2.1.1 Downloading

SST/macro is available at `https://github.com/sstsimulator/sst-macro`

```
shell> git clone https://github.com/sstsimulator/sst-macro.git
```

or for ssh

```
shell> git clone ssh://git@github.com/sstsimulator/sst-macro.git
```

If you'd like to use ssh for convenience, make sure you have added a public key for your GitHub account.

2.1.2 Dependencies

The most critical change is that C++11 is now a strict prerequisite. Workarounds had previously existed for older compilers. These are no longer supported. The following are dependencies for SST/macro.

- (optional) Git 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: 2.68 or later
- Automake: 1.11.1 or later
- Libtool: 2.4 or later
- A C/C++ compiler is required with C++11 support. gcc 4.8 and onward is known to work.
- (optional) Doxygen and Graphviz are needed to build the documentation.
- (optional) Graphviz is needed to collect call graphs.

2.1.3 Configuration and Building

SST/macro is an SST element library, proving a set of simulation components that run on the main SST core. The SST core provides the parallel discrete event simulation manager that manages time synchronization and sending events in serial, MPI parallel, multi-threaded, or MPI + threaded mode. The core does not provide any simulation components like node models, interconnect models, MPI trace readers, etc. The actual simulation models are contained in the element library.

The SST core is a standalone executable that dynamically loads shared object files containing the element libraries. For many element libraries, a Python input file is created that builds and connects the various simulation components. For maximum flexibility, this will become the preferred mode. However, SST/macro has historically had a text-file input `parameters.ini` that configures the simulation. To preserve that mode for existing users, a wrapper Python script is provided that processes SST/macro input files.

The workflow for installing and running is therefore:

- Build and install SST core

- Build and install the SST/macro element library `libmacro.so`

- Make sure paths are properly configured for `libmacro.so` to be visible to the SST core

- Run the `sstmac` wrapper Python script that runs SST/macro-specific parameters OR

- Write a custom Python script

Build SST core

The recommended mode for maximum flexibility is to run using the SST core downloadable from `http://sst-simulator.org/SSTPages/SSTMainDownloads/`. Building and installing sets up the discrete event simulation core required for all SST elements. SST core still has a few Boost dependencies, which should be the only complication in building. For building Boost, we recommend two files: `user-config.jam` to configure the Boost compiler flags and a `runme.sh` that bootstraps, compiles, and installs the prerequisite Boost libraries. For GCC, the `user-config.jam` should go in the top-level home directory and the file should contain the line:

```
using gcc : : $PATH_TO_MPIC++  : <compileflags>-std=c++1y ;
```

For Clang on Mac, use:

```
using clang : : $PATH_TO_CLANG_MPIC++  : <compileflags>-std=c++1y <linkflags>-stdlib=libc++
```

We recommend Boost 1.59. Other Boost versions should work as well, but this seems the most stable. In the top-level Boost directory, make a script `runme.sh` that contains:

```
./bootstrap.sh \
  --with-libraries=program_options,serialization,filesystem \
  --with-toolset=gcc

./b2 ---prefix=$INSTALL
./b2 --layout=tagged --prefix=$INSTALL
```

The toolset can be changed from gcc to clang, as needed. For maximum safety, Boost should install both "tagged" versions of libraries and un-tagged versions. Once Boost is installed with these options, configuration and installation of SST core should be straightforward following documentation in the core library.

Build SST/macro element library

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

```
sst-macro> ./bootstrap.sh
sst-macro> mkdir build
sst-macro> cd build
sst-macro/build> ../configure --prefix=/path-to-install --with-sst-core=$PATH_TO_SST_CORE CC=$MPICC CXX=$MPICXX
```

`PATH_TO_SST_CORE` should be the prefix install directory used when installing the core. The MPI compilers should be the same compilers used for building Boost and SST core.

SST/macro can still be built in standalone mode for a select set of features that have not been fully migrated to the SST core. The installation and running are the same for both modes - simply remove the `--with--sst` parameter. A complete list of options for standalone building can be seen by running `../configure --help`. Some common options include:

- –(dis|en)able-graphviz : Enables the collection of simulated call graphs, which can be viewed with graphviz. Enabled by default. Disable if not using Boost or C++11. Ordered maps can be used as a replacement, but with lower performance.

- –(dis|en)able-regex : Regular expressions can be used to proofread input files, but this requires either Boost or C++11. Enabled by default. Disable if not using Boost or C++11.

- –(dis|en)able-custom-new : Memory is allocated in larger chunks in the simulator, which can speed up large simulations.

- –(dis|en)able-multithread : This configures for thread-level parallelism for (hopefully) faster simulation

Once configuration has completed, printing a summary of the things it found, simply type `make`.

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:

```
sst-macro/build> make -j8 check
sst-macro/build> make install
sst-macro/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: Applications and other code linking to SST/macro use Makefiles that use the sst++/sstcc compiler wrappers that are installed there for convenience to figure out where headers and libraries are. Make sure your path is properly configured.

2.1.5 Known Issues

- Any build or runtime problems should be reported to sstmacro-devel@googlegroups.com. We try to respond as quickly as possible.

- make -j: When doing a parallel compile dependency problems can occur. There are a lot of inter-related libraries and files. Sometimes the Makefile dependency tracker gets ahead of itself and you will get errors about missing libraries and header files. If this occurs, restart the compilation. If the error vanishes, it was a parallel dependency problem. If the error persists, then it's a real bug.

- Ubuntu: The Ubuntu linker performs too much optimization on dynamically linked executables. Some call it a feature. I call it a bug. In the process it throws away symbols it actually needs later. When compiling with Ubuntu, make sure that '-Wl,–no-as-needed' is always given in LDFLAGS.

    The problem occurs when the executable depends on libA which depends on libB. The executable has no direct dependence on any symbols in libB. Even if you add `-lB` to the `LDFLAGS` or `LDADD` variables, the linker ignores them and throws the library out. It takes a dirty hack to force the linkage. If there are issues, contact the developers at sstmacro-devel@googlegroups.com and report the problem. It can be fixed easily enough.

- Compilation with clang should work, although the compiler is very sensitive. In particular, template code which is correct and compiles on several other platforms can mysteriously fail. Tread with caution.

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 `sst-macro/sst-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 `https://github.com/sstsimulator/sst-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) :

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

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 Running an Application

2.3.1 SST Python Scripts

Full details on building SST Python scripts can be found at `http://sst-simulator.org`. To preserve the old parameter format in the near-term, SST/macro provides the `pysstmac` script:

```
export SST_LIB_PATH=$SST_LIB_PATH:$SSTMAC_PREFIX/lib

options="$@"
$SST_PREFIX/bin/sst $SSTMAC_PREFIX/include/python/default.py --model-options="$options"
```

The script configures the necessary paths and then launches with a Python script `default.py`. Interested users can look at the details of the Python file to understand how SST/macro converts parameter files into a Python config graph compatible with SST core. Assuming the path is configured properly, users can run

```
>pysstmac -f parameters.ini
```

with a properly formatted parameter file. If running in standalone mode, the command would be similarly (but different).

```
from sst.macro import *
setupDeprecated()
```

```
>sstmac -f parameters.ini
```

since there is no Python setup involved.

2.3.2 Building Skeleton Applications

To demonstrate how an external skeleton application is run in SST/macro, we'll use a very simple send-recv program located in `skeletons/sendrecv`. We will take a closer look at the actual code in Section 3.5. After SST/macro has been installed and your PATH variable set correctly, for standalone core users can run:

```
sst-macro> cd skeletons/sendrecv
sst-macro/skeletons/sendrecv> make
sst-macro/skeleton/sendrecv> ./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. For running on the main SST core, a few extra flags are required. Rather than generating a standalone executable, the compiler wrapper generates a shared library. Users can always write their own Python scripts, which will be required for more advanced usage. However, users can also just use the `pysstmac` script.

```
>sst-macro/skeletons/sendrecv> pysstmac librunsstmac.so -f parameters.ini
```

Any extra shared libraries can be given as the first few parameters and these will automatically be imported.

2.3.3 Makefiles

We recommend structuring the Makefile for your project like the one seen in `skeletons/sendrecv/Makefile` :

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

CXX :=        $(PATH_TO_SST)/bin/sst++
CC :=          $(PATH_TO_SST)/bin/sstcc
CXXFLAGS := ...
CPPFLAGS := ...
LIBDIR :=   ...
PREFIX :=     ...
LDFLAGS :=   -Wl,-rpath,$(PREFIX)/lib
...
```

The SST compiler wrappers `sst++` and `sstcc` automagically configure and map the code for simulation. More details are given in Section 5.2. If using external skeleton applications, the default Python script for SST/macro will not work and a small modification will be required.

2.3.4 Command-line arguments

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

- -h/–help: 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).

- –dumpi: If you are in a folder with all the DUMPI traces, you can invoke the main `sstmac` executable with this option. This replays the trace in a special debug mode for quickly validating the correctness of a trace.

- -d [debug flags]: A list of debug flags to activate as a comma-separated list (no spaces) - see Section 2.5

- -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 multithreaded, give a comma-separated list (no spaces) of the core affinities to use - see Section 2.4.2

2.4 Parallel Simulations in Standalone Mode

SST/macro supports running parallel discrete event simulation (PDES) in distributed memory (MPI), threaded shared memory (pthreads) and hybrid (MPI+pthreads) modes. Running these in standalone mode is generally discouraged as parallel simulations should use the unified SST core.

2.4.1 Distributed Memory Parallel

Configure will automatically check for MPI. Your configure should look something like:

```
sst-macro/build> ../configure CXX=mpicxx CC=mpicc ...
```

With the above options, you can just compile and go. SST/macro is run exactly like the serial version, but is spawned like any other MPI parallel program. Use your favorite MPI launcher to run, e.g. for OpenMPI

```
mysim> mpirun -n 4 sstmac -f parameters.ini
```

or for MPICH

```
mysim> mpiexec -n 4 sstmac -f parameters.ini
```

Even if you compile for MPI parallelism, the code can still be run in serial with the same configuration options. SST/macro will notice the total number of ranks is 1 and ignore any parallel options. When launched with multiple MPI ranks, SST/macro will automatically figure out how many partitions (MPI processes) you are using, partition the network topology into contiguous blocks, and start running in parallel.

2.4.2 Shared Memory Parallel

In order to run shared memory parallel, you must configure the simulator with the `--enable-multithread` flag. Partitioning for threads is currently always done using block partitioning and there is no need to set an input parameter. Including the integer parameter `sst_nthread` specifies the number of threads to be used (per rank in MPI+pthreads mode) in the simulation. The following configuration options may provide better threaded performance.

- `--enable-spinlock` replaces pthread mutexes with spinlocks. Higher performance and recommended when supported.

- `--enable-cpu-affinity` causes SST/macro to pin threads to specific cpu cores. When enabled, SST/macro will require the `cpu_affinity` parameter, which is a comma separated list of cpu affinities for each MPI task on a node. SST/macro will sequentially pin each thread spawned by a task to the next next higher core number. For example, with two MPI tasks per node and four threads per MPI task, `cpu_affinity = 0,4` will result in MPI tasks pinned to cores 0 and 4, with pthreads pinned to cores 1-3 and 5-7. For a threaded only simulation `cpu_affinity = 4` would pin the main process to core 4 and any threads to cores 5 and up. The affinities can also be specified on the command line using the `-c` option. Job launchers may in some cases provide duplicate functionality and either method can be used.

2.4.3 Warnings for Parallel Simulation

- Watch your `LD_LIBRARY_PATH` if you have multiple different builds. If your paths get scrambled and the wrong libraries are being read, you will get bizarre, inscrutable errors.

- If the number of simulated processes specified by e.g. `aprun -n 100` does not match the number of nodes in the topology (i.e. you are not space-filling the whole simulated machine), parallel performance will suffer. SST/macro partitions nodes, not processes.

- If running an MPI program, you should probably be safe and use the `mpi_check` debug flag (see below) to ensure the simulation runs to completion. The flag ensures `MPI_Finalize` is called and the simulation did not "deadlock." While the PDES implementation should be stable, it's best to treat it as Beta++ to ensure program correctness.

> Parallel simulation speedups are likely to be modest for small runs. Speeds are best with serious congestion or heavy interconnect traffic. Weak scaling is usually achievable with 100-500 simulated MPI ranks per logical process. Even without speedup, parallel simulation can certainly be useful in overcoming memory constraints, expanding the maximum memory footprint.

2.5 Debug Output

SST/macro defines a set of debug flags that can be specified in the parameter file to control debug output printed by the simulator. To list the set of all valid flags with documentation, the user can run

```
bin> ./sstmac --debug-flags
```

which will output something like

```
mpi
    print all the basic operations that occur on each rank - only API calls are
    logged, not any implementation details
mpi_check
    validation flag that performs various sanity checks to ensure MPI application
    runs and terminates cleanly
mpi_collective
    print information about MPI collective calls as well as implementation details
mpi_pt2pt
    print information about MPI point-to-point calls as well as implementation
    details
 ....
```

The most important flag for validating simulations is the `mpi_check` flag, which causes special sanity checks and a final validation check to ensure the simulation has finished cleanly. Some of the debug flags can generate information overload and will only be useful to a serious developer, rather than a user.

To turn on debug output, add the following to the input file

```
debug = mpi   mpi_check
```

listing all flags you want separated by spaces. Note: this is a major shift from the previous (and really tedious, unfriendly) debug system of past versions. The new system allows much finer-grained, simpler printing of debug output. Additionally, it allows new debug flags to very easily defined. More info on declaring new debug flags in your own code can be found in the developer's reference.

Chapter 3

Basic Tutorials

3.1   SST/macro Parameter files

A minimal parameter file setting up a 2D-torus topology is shown below. A detailed listing of parameter namespaces and keywords is given in Section 6. The preferred input files now use namespaces. However, for consistency with previous versions, we also show the deprecated parameters.

```
# Launch parameters
app1.indexing = block
app1.allocation = first_available
app1.launch_cmd = aprun -n8 -N1
app1.name = sstmac_mpi_testall
app1.argv =
# Application parameters
app1.sendrecv_message_size = 128
# Network parameters
amm_model = amm1
congestion_model = simple
switch.ejection.bandwidth = 1GB/s
switch.ejection.latency = 100ns
switch.link.bandwidth = 1.0GB/s
switch.link.latency = 100ns
# Topology - 4x4 Torus
topology.name = torus
topology.geometry = 4,4
# Node parameters
node.ncores = 1
node.model = simple
node.memory.model = simple
node.memory.bandwidth = 1GB/s
node.memory.latency = 10ns
node.proc.frequency = 1GHz
nic.model = simple
nic.injection.bandwidth = 1GB/s
nic.injection.latency = 1us
```

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 #.

3.1.1 Parameter Namespace Rules

Periods denote nesting of parameter namespaces. The parameter `node.memory.model` will be nested in namespace `memory` inside namespace `node`. SST/macro generally requests variable values via the inner-most namespace only. This means rather than asking for variable `node::memory::model` it will actually look for variable `memory::model` starting with the most deeply nested namespace. This exactly follows C++ namespace rules.

For example, consider the following:

```
memory.model = simple
node.memory.model = pisces
```

If I am building the node's memory system, the initialization will look for `memory::model` inside namespace `node` first, returning the value `pisces`. If node initialization had been unable to find the variable in the `node` namespace, it would have moved up and looked in the global namespace, returning the value `simple`.

A new syntax supported in 6.1 more closely resembles C++ namespace declarations. Namespaces can be scoped using brackets {}:

```
node {
 model = simple
 memory {
   model = simple
   bandwidth = 1GB/s
   latency = 10ns
 }
}
```

Any line containing a single string with an opening { starts a new namespace. A line containing only a closing } ends the innermost namespace. The syntax is not as flexible as C++ since the opening { must appear on the same line as the namespace and the closing } must be on a line of its own.

Using the new syntax, the parameter file above now becomes:

```
amm_model = amm1
congestion_model = simple

app1 {
 indexing = block
 allocation = first_available
 launch_cmd = aprun −n8 −N1
 name = sstmac_mpi_testall
 argv =
 sendrecv_message_size = 128
}

switch {
 ejection {
   bandwidth = 1GB/s
   latency = 100ns
 }
 link {
   bandwidth = 1.0GB/s
   latency = 100ns
 }
}

topology {
 name = torus
 geometry = 4,4
}
```

```
node {
 ncores = 1
 model = simple
 memory {
   bandwidth = 1GB/s
   latency = 10ns
   model = simple
 }
 proc.frequency = 1GHz
 nic {
   injection.bandwidth = 1GB/s
   injection.latency = 1us
 }
}
```

Again, a detailed listing of parameter namespaces and keywords is given in Section 6.

3.1.2 Initial Example

Continuing with the example above, we see 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 `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 3.4 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
app1.indexing = block
app1.allocation = first_available
app1.launch_cmd = aprun -n2 -N1
app1.name = user_mpiapp_cxx
app1.argv =
# Application parameters
app1.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`.

Using the deprecated (non-namespace) parameters the file would be:

```
# Launch parameters
launch_indexing = block
launch_allocation = first_available
launch_app1_cmd = aprun −n2 −N1
launch_app1 = user_mpiapp_cxx
launch_app1_argv =
# Network parameters
network_name = simple
network_bandwidth = 1.0GB/s
network_hop_latency = 100ns
# Topology − Ring of 4 nodes
topology_name = torus
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
```

3.2   Abstract Machine Models

The preferred mode for usage of SST/macro will be through specifying parameters for well-defined abstract machine models. This represents an intermediate-level mode that should cover the vast majority of use cases. The highly configurable, detailed parameter files will remain valid but will represent advanced usage mode for corner cases. The primary advantage of the abstract machine models is a uniform set of parameters regardless of the underlying congestion model or accuracy level (Pisces or LogGOPSim). Each input file requires the usual set of software parameters given in 3.1. For hardware parameters, two initial parameters are required and one is optional.

```
congestion_model = pisces
amm_model = amm1
accuracy_parameter = 1024
```

Here we indicate the congestion model to be used (the packet-flow) and the overall machine model (abstract machine model #1). Currently valid values for the congestion model are `pisces` (most accurate, slowest) and `simple` (least accurate, fastest), but more congestion models should be supported in future versions. Currently valid values for the abstract machine model are `amm1`, `amm2`, `amm3`, see details below. Another model, `amm4`, that adds extra detail to the NIC is pending and should be available soon. The details of individual abstract machine models are given in the following sections. The optional accuracy parameter is less well-defined and the exact meaning varies considerably between congestion models. In general, the accuracy parameter represents how coarse-grained the simulation is in bytes. It basically corresponds to a packet-size. How many bytes are modeled moving through the machine separately at a time? If the parameter is set to 8 bytes, e.g., that basically means we are doing flit-level modeling. If the parameter is set to 8192 bytes, e.g. that means we are doing very coarse-grained modeling which only really affects large messages. If the parameter is set to 100-1000 bytes, e.g., that means we are doing more fine-grained modeling on real packet sizes, but we are ignoring flit-level details.

3.2.1   Common Parameters

The following parameters define the CPU and compute power of the node (independent of memory subsystem). They are universal and are valid for all abstract machine models.
Using the preferred (current) namespace parameters:

```
node.model = simple
node.frequency = 2.1ghz
node.ncores = 24
node.nsockets = 4
```

or using the deprecated parameters:

```
node_name = simple
node_frequency = 2.1ghz
node_ccores = 24
node_sockets = 4
```

3.2.2   AMM1
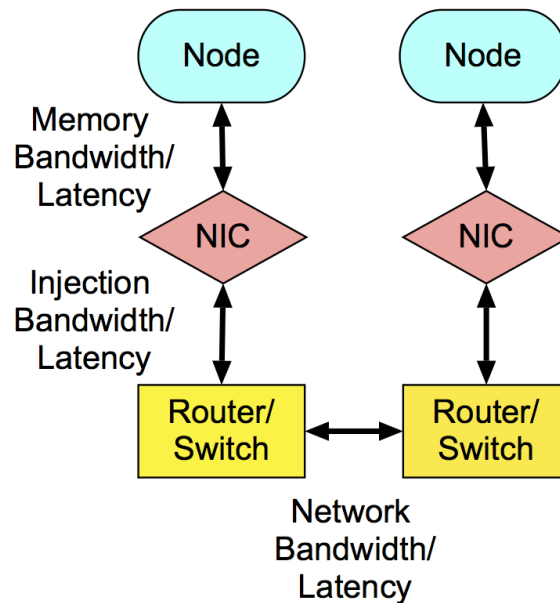
Figure 3.1: AMM1: Used when focusing on network traffic only

This is simplest abstract machine model and incorporates three basic components (i.e. congestion points). Each node has a memory subsystem and NIC (injection/ejection). Once packets are injected, they traverse a series of network switches. The memory, injection, and network are all defined by a bandwidth/latency parameter pair.

```
network_link_bandwidth = 6GB/s
network_hop_latency = 100ns
injection_bandwidth = 10GB/s
injection_latency = 1us
memory_bandwidth = 10GB/s
memory_latency = 15ns
```

These are special parameters used by the AMM configurations. They can be by-passed by directly using fully namespaced parameters:

```
switch {
 link {
   bandwidth = 6GB/s
   latency = 100ns
 }
 ejection {
  latency = 100ns
  bandwidth = 10GB/s
 }
}
nic {
 injection {
  latency = 1us
  bandwidth = 10GB/s
 }
}
memory {
 bandwidth = 10GB/s
 latency = 15ns
}
```

NOTE: there is no parameter `network_latency`. The parameter is `network_hop_latency`. This is the latency required for a single packet to traverse one switch and hop to the next one in the network. Thus, even in the most basic of network models, there is a still a notion of topology that affects the number of hops and therefore the latency. To compute the total network network latency as one would observe in an MPI ping-ping benchmark, one would compute

$$lat = n_{hops} * lat_{hop} + 2 * lat_{inj}$$

using the hop latency and the injection latency.

This abstract machine model is a good place to start for getting a "lay of the land" for simulations - and the simplest to configure. However, it has a few deficiencies that can cause problems when there is serious memory or network congestion. More details (and their fixes) are given in the next abstract machine models.

3.2.3   AMM2

A major deficiency of AMM1 is that it grants exclusive access to memory resources. Two CPUs or the NIC cannot be using the memory subsystem in parallel. This is particularly problematic for large memory transfers (1 MB or greater). The memory system might be blocked for approx 1 ms, creating unphysical delays while other resources wait for access. A more realistic model allows multiple resources to access the memory, albeit with reduced bandwidth when congestion is observed. In many cases, multiple memory links or management units are connect to a shared bus. The bus determines to the total, aggregate memory bandwidth. However, the individual links determine the maximum observed bandwidth by any single component. AMM2 has all the same parameters as AMM1, but now allows an additional parameter for memory. These are special parameters used by the AMM configurations. They can be by-passed by directly using fully namespaced parameters (not shown).

```
max_memory_bandwidth = 5GB/s
memory_bandwidth = 10GB/s
memory_latency = 15ns
```

The new parameter `max_memory_bandwidth` now defines the maximum bandwidth any single component is allowed. Thus, even if the CPU is doing something memory intensive, 5 GB/s is still available to the NIC for network transfers. We remark here that the memory parameters might be named something slightly more descriptive. However, as a rule, we want the AMM1 parameters to be a proper subset of the AMM2 parameters. Thus parameter names should not change - only new parameters should be added.
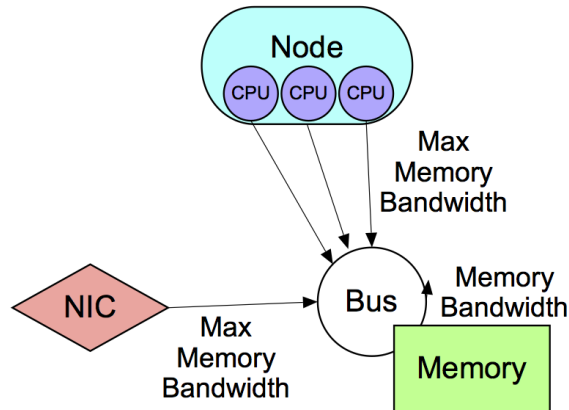
Figure 3.2: AMM2: Adds extra memory model details to AMM1

3.2.4 AMM3

A major deficiency of AMM2 is its inability to distinguish between the network link bandwidth (associated with the outport port serializer/deserializer) and the switch bandwidth (associated with the crossbar that arbitrates packets). Only packets traveling the same path cause congestion on the network links in AMM1 and AMM2. However, packets "intersecting" at a switch - even if following separate paths - can cause congestion through sharing the switching fabric. AMM3 generalize the network parameters by adding a switch bandwidth. We note again here that AMM3 has all the same parameters as AMM2, plus the additional switch bandwidth parameter. Thus, higher-numbered abstract machine models always add more detail. These are special parameters used by the AMM configurations. They can be by-passed by directly using fully namespaced parameters (not shown) for more detailed configurations.

```
network_switch_bandwidth = 12GB/s
network_bandwidth = 6GB/s
network_hop_latency = 100ns
```

3.3 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.

3.3.1 Topology

Topologies are determined by two mandatory parameters.

```
topology.name = torus
topology.geometry = 4 4
```
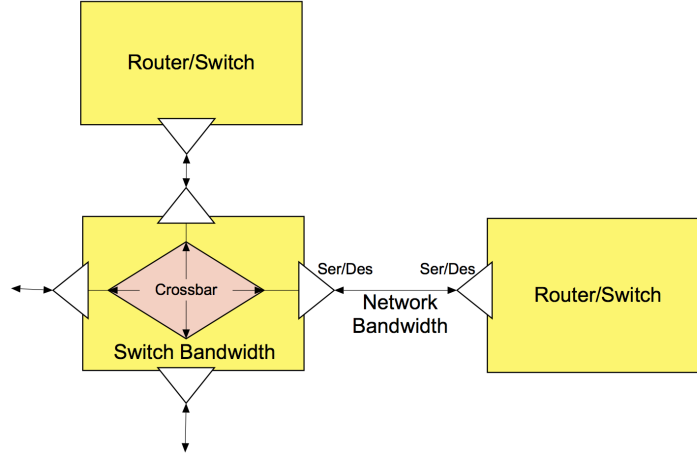
Figure 3.3: AMM3: Adds extra router (switch) details to AMM2

Here we choose a 2D-torus topology with extent 4 in both the X and Y dimensions for a total of 16 nodes (Figure 3.4) 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 3.5. 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
topology.concentration = 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

```
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
```
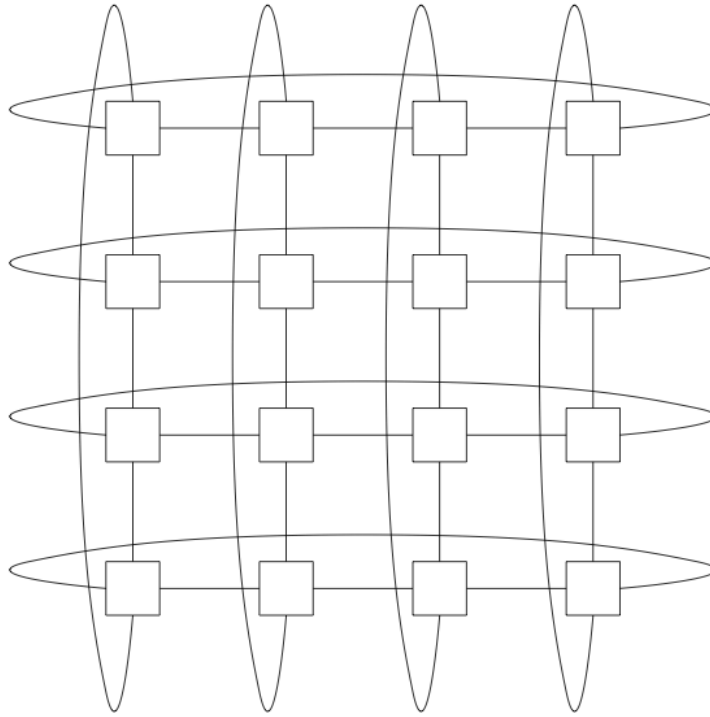
Figure 3.4: 4 x 4 2D Torus

```
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.

3.3.2   Routing

By default, SST/macro uses the simplest possible routing algorithm: dimension-order minimal routing (Figure 3.6). In going from source to destination, the message first travels along the X-dimension and then travels 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:

```
router = min_ad
```

which specifies minimal adaptive routing. There are now multiple valid paths between network endpoints, one of which is illustrated in Figure 3.7. At each network hop, the router chooses the *productive* path with least congestion. In some cases,
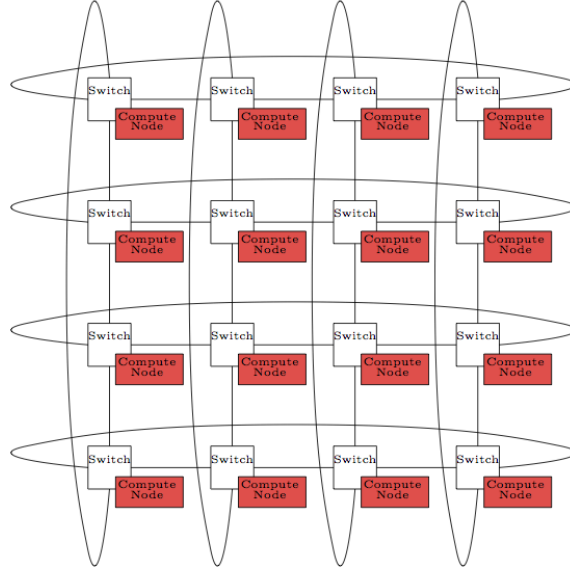
Figure 3.5: 4 x 4 2D Torus of Network Switches with Compute Nodes

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.

3.4 Network Model

Network models can be divided into several categories. SST/macro supports analytic models, which estimate network delays via basic latency/bandwidth formulas, and packet models, which model step-by-step the transit of individuals through the interconnect. A third class of models (flow models), was previously supported but are now discontinued due to the much better scalability of packet models.

3.4.1 Analytic Models: MACRELS

The analytic models in SST/macro are colloqiually referred to as MACRELS (MTL for AnalytiC REally Lightweight Simulation). The MTL (message transfer layer) moves entire network flows from point-to-point without packetizing them into smaller chunks. Thus an entire 1 MB MPI message is transported as a single chunk of data. The majority of MACRELS models are based on the LogP set of approximations:

$$\Delta t = \alpha + \beta N$$

Figure 3.6: Dimension-Order Minimal Routing on a 2D Torus

Figure 3.7: Adaptive Minimal Routing on a 2D Torus

where Δt is the time delay, α is the minimum latency of the communication, β is the inverse bandwidth (s/B), and N is the number of bytes. In abstract machine models, these methods are selected as:

```
congestion_model = logP
```

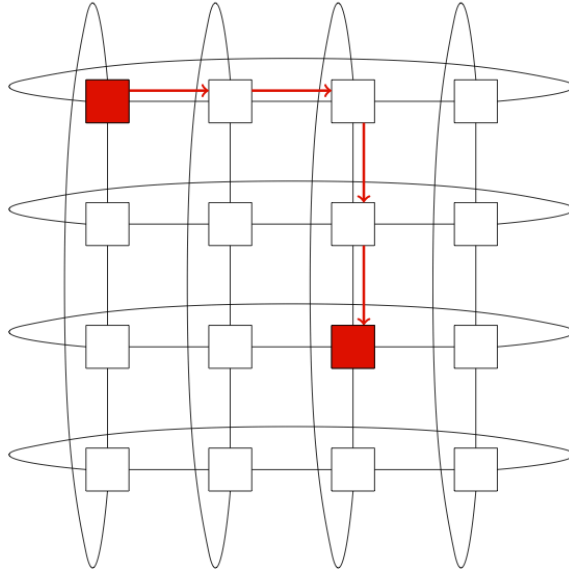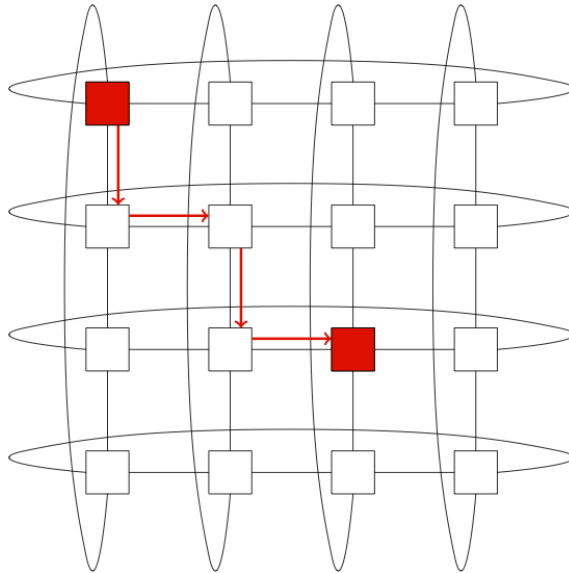### 3.4.2 Packet Models: PISCES

PISCES (Packet-flow Interconnect Simulation for Congestion at Extreme Scale) breaks network flows (MPI messages) into individual packets and models each packet individually. In abstract machine models, PISCES can be selected as:

```
congestion_model = pisces
```

In reality, packets are further subdivided into flits (flow-control units). Flit-level detail would be way too computationally intense for large-scale simulation. All routing decisions are made on packets as a while. Two flits in the same packet cannot take different paths through the network. However, they may not travel together. PISCES provides two mechanisms for treating flit-level flow control.

PISCES simple model

In the simple model, each router uses a basic store-and-forward mechanism. Flits are not allowed to "separate" and always travel as a single unit. The entire packet has to be stored within a router before it can be forwarded to the next router. The simple model affects the arbitrator that decided when and how to transmit flits. To select a simple model:

```
arbitrator = simple
```

The simple model is the least computationally expensive. However, for large packet sizes, it can produce erroneously high latencies. To tune the packet size for abstract machine models, set:

```
accuracy_parameter = 1024B
```

or equivalently

```
mtu = 1024B
```

which sets the packet size to 1024B. For the simple model, packet sizes larger than 256-512B are not recommended. Packet sizes on production supercomputers are often small (96-128B). Small packet sizes with the simple model can be a good compromise for having more fine-grained routing but cheaper congestion modeling in the arbitrator.

PISCES cut-through model

In the cut-through model, routing decisions still occur at the packet-level. However, some attempt is made to account for pipelining of flits across different router stages. Somewhat similar to the LogP models used above, latency/bandwidth formulas are used to estimate packet delays. However, the cut-through model adds more details. It's requested as:

```
arbitrator = cut_through
```

Figure 3.8 shows a timeline for the data being transmitted through a crossbar, SerDes, or other network component with a "fixed bandwidth." Each component is essentially a pipe with some flow bandwidth. The arbitrator divides its limited bandwidth amongst incoming packets. Packets fill the pipeline, consuming bandwidth. In contrast to the completely discrete simple model, packets can "multiplex" in the component sharing an arbitrary bandwidth partition. Modeling a packet delay starts with two input parameters and computes three output parameters.
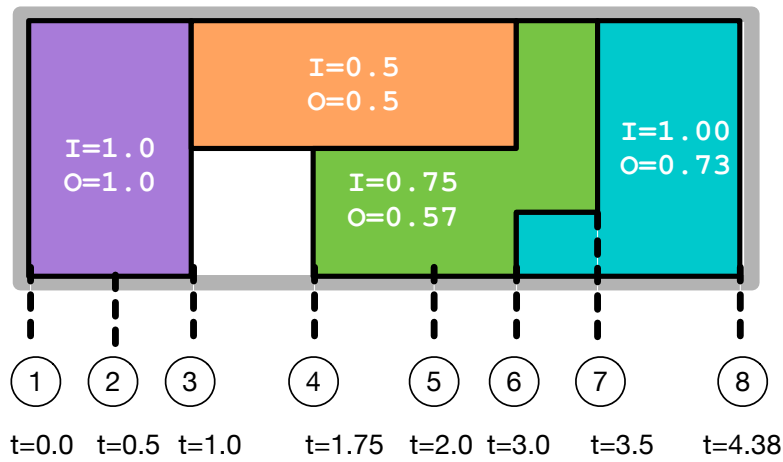
Figure 3.8: Timeline of four different packets passing through a PISCES cut-through bandwidth arbitrator. The incoming bandwidth (I) and outgoing bandwidth (O) are shown for each packet. Time is the horizontal axis. Bandwidth consumed by a packet is shown by the vertical extent of each packet. The individual events are 1) First packet arrives 2) Second packet arrives with reduced bandwidth but no available bandwidth 3) First packet finishes. Second packet can begin sending. 4) Third packet arrives and begins sending with remaining bandwidth. 5) Fourth packet arrives, but no available bandwidth. 6) Second packet finishes. Third packet increases bandwidth. Fourth packet can begin sending. 7) Third packet finishes. Fourth packet increases bandwidth. 8) Fourth packet finishes. Full details are given in the text.

- A: Packet head arrival (input)

- I: Packet incoming bandwidth (input)

- H: Packet head departure (output)

- T: Packet tail departure (output)

- O: Packet outgoing bandwidth (output)

In the simple model, a packet either consumes all the bandwidth or none of the bandwidth. To account for flit-level pipelining, the cut-through model allows packets to consume partial bandwidths. Consider an aribitrator that has a maximum bandwidth of 1.0. The first packet (purple, Figure 3.8) arrives with a full incoming bandwidth of 1.0 and head arrival of t=0.0. It therefore consumes all the available bandwidth. The head of the packet can actually leave immediately (as it must to properly pipeline or cut-through). The tail leaves after all bytes have sent at t=1.0. Thus for the first packet we have H=0.0, T=1.0, and O=1.0.

The second packet (orange) arrives at t=0.5. Upon arrival there is no bandwidth available as the first packet is consuming the maximum. Only after the first packet finishes can the second packet begin. The second packet arrives and leaves with a reduced bandwidth of 0.5. Thus we have H=1.0, T=3.0, and O=0.5.

The third packet (green) arrives at t=1.75. Upon arrival there is some bandwidth, but not enough to match the incoming bandwidth of 0.57. Thus the third packet is slowed initially. After the second packet finished, the third packet can send at increased bandwidth. The computation here is a bit more complicated. Packet 3 can actually consume MORE than 0.6 bandwidth units. Between steps 4 and 6, packet 3 has accumulated in a local buffer in the router. Thus even though the incoming bandwidth is only 0.6, there are several flits that are available to send immediately at full bandwidth waiting in the buffer. Thus results in an effective bandwidth of 0.75 for the remainder of the packet's time in the arbitrator. Thus we end up with H=1.75, T=3.5, and O=0.57. Even though the packet is initially delayed, the buffers compensate for the delay and allow the outgoing bandwidth to "catch up" with the incoming bandwidth.

Finally, the fourth packet (blue) arrives at t=3.0. There is some available bandwidth. After the third packet finishes, the fourth packet can now send at maximum. Because of the initial delay, the outgoing bandwidth is somewhat reduced. We have H=3.0, T=4.38, and O=0.73.

3.4.3 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. Congestion is treated as a fluid dynamics problem, sharing bandwidth between competing flows. In contrast to LogP models, flow models can account fairly well for congestion. Without congestion, a flow only requires a FLOW START and FLOW STOP event to be modeled (see tutorial on discrete event simulation in 3.7). 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 PISCES simple 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. Support for this model has been completely removed.

3.5 Basic MPI Program

Let us go back to the simple send/recv skeleton and actually look at the code. This code should be compiled with SST compiler wrappers installed in the `bin` folder.

```
1   #include <stdlib.h>
2   #include <stdio.h>
3   #include <mpi.h>
4
5   #define sstmac_app_name simple_test
6
7   int main(int argc, char **argv)
8   {
9     int message_size = 128;
10    int me, nproc;
11    int tag = 0;
12    int dst = 1;
13    int src = 0;
14    MPI_Status stat;
15
16    MPI_Init(&argc,&argv);
17    MPI_Comm world = MPI_COMM_WORLD;
18    MPI_Comm_rank(world,&me);
19    MPI_Comm_size(world,&nproc);
```

The starting point is creating a main routine for the application. The simulator itself already provides a `main` routine. The SST compiler automatically changes the function name to `user_skeleton_main`, which provides an entry point for the application to actually begin. When SST/macro launches, it will invoke this routine and pass in any command line arguments specified via the `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

```
app1.name = simple_test
```

The name associated to the application is given by the `sstmac_app_name` macro. This macro must be defined to a unique string name in the source file containing `main`. SST/macro will automatically associate the given main routine with the string internally. That application can then be selected in the input file with `app1.name`.

At the very top of the file, the `mpi.h` header is actually mapped by the SST compiler to an SST/macro header file. This header provides the MPI API and configures MPI function calls to link to SST/macro instead of the real MPI library. The code now proceeds:

```
1     if (nproc != 2) {
2       fprintf(stderr, "sendrecv only runs with two processors\n");
3         abort();
4     }
5     if (me == 0) {
6       MPI_Send(NULL, message_size, MPI_INT, dst, tag, world);
7       printf("rank %i sending a message\n", me);
8     }
9     else {
10      MPI_Recv(NULL, message_size, MPI_INT, src, tag, world, &stat);
11      printf("rank %i receiving a message\n", me);
12    }
13    MPI_Finalize();
14    return 0;
15  }
```

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

3.6 Launching, Allocation, and Indexing

3.6.1 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.

```
app1.name = user_mpiapp_cxx
app1.launch_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.

3.6.2 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
topology.concentration = 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 3.9)

```
app1.allocation = first_available
```
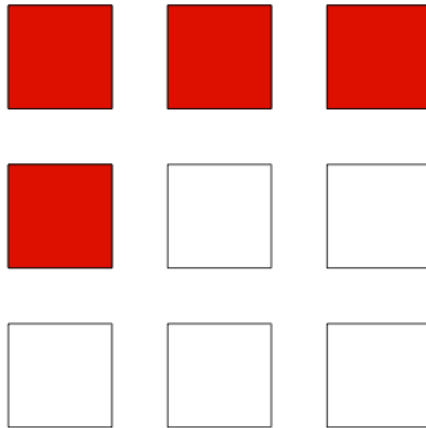
Figure 3.9: First available Allocation of 4 Compute Codes on a 3x3 2D Torus

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 3.10).

```
app1.allocation = cartesian
app1.cart_sizes = 2 2
app1.cart_offsets = 0 0
```
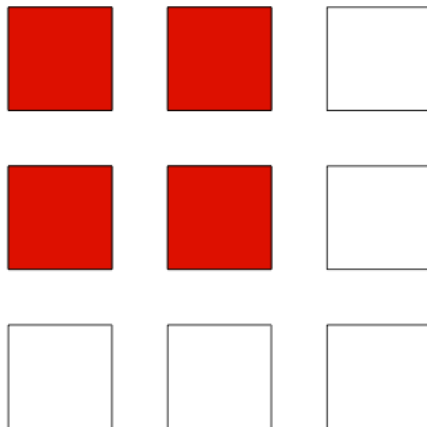

Figure 3.10: Cartesian Allocation of 4 Compute Codes on a 3x3 2D Torus

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.

```
app1.allocation = random
```

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 3.12).

```
app1.indexing = block
```

In contrast, round-robin spreads out MPI ranks by assigning consecutive MPI ranks on different nodes (Figure 3.13).
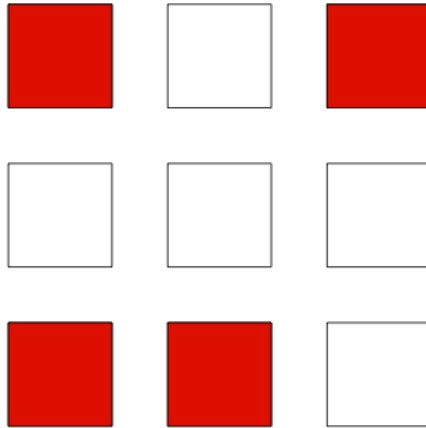
Figure 3.11: Random Allocation of 4 Compute Codes on a 3x3 2D Torus

```
app1.indexing = round_robin
```

Finally, one may also choose

```
app1.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.

3.7   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 3.14). In the most common (and simplest) use case, all user-space threads are serialized, running one at a time. 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.

Figure 3.12: Block Indexing of 8 MPI Ranks on 4 Compute Nodes

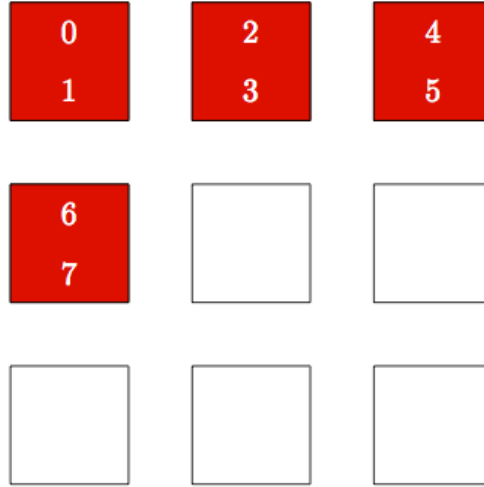The simulation begins at time $t = 0\mu s$. The simulation thread runs the first event, launching process 0. The context of process 0 is switched in, and SST/macro proceeds running code as if it were actually process 0. Process 0 starts a blocking send in Event 1. For process 0 to perform a send in the 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.
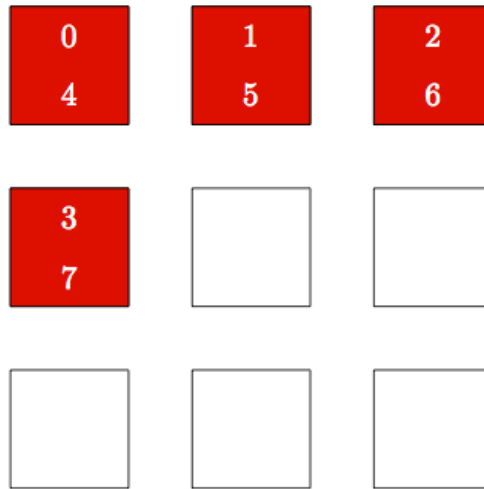
Figure 3.13: Round-Robin Indexing of 8 MPI Ranks on 4 Compute Nodes

	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 (1μs)	5)Wait for ack; block	
$t = 2\mu s$	7)Recv at NIC 1; unblock proc 1		8)Send ack for recv (1μs); block
$t = 3\mu s$	9)Deliver ack to NIC 0 (1μs) 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 3.14: Progression of Discrete Event Simulation for Simple Send/Recv Example

3.8 Using DUMPI

3.8.1 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 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/sst-ca/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

```
1  int MPI_Send(...)
2  {
3    /** Start profiling work */
4    ...
5    int rc = PMPI_Send(...);
6    /** Finish profiling work */
7    ...
8    return rc;
9  }
```

collecting profile information and then directly calling the PMPI functions.

We examine DUMPI using a very basic example program.

```
1  #include <mpi.h>
2  int main(int argc, char** argv)
3  {
4    MPI_Init(&argc, &argv);
5    MPI_Finalize();
6    return 0;
7  }
```

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 used 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

It is often useful to validate the correctness of a trace. Sometimes there can be problems with trace collection. There are also a few nooks and crannies of the MPI standard left unimplemented. To validate the trace, you can run in a special debug mode that runs the simulation with a very coarse-grained model to ensure as quickly as possible that all functions execute correctly. This can be done straightforwardly by running the executable with the dumpi flag: `sstmac --dumpi`.

To replay a trace in the simulator, a small modification is required to the example input file in 3.1. We have two choices for the trace replay. First, we can attempt to *exactly* replay the trace as it ran on the host machine. Second, we could replay the trace on a new machine or different layout.

For exact replay, the key issue is specifying the machine topology. For some architectures, topology information can be directly encoded into the trace. This is generally true on Blue Gene, but not Cray. When topology information is recorded, trace replay is much easier. The parameter file then becomes, e.g.

```
app1.launch_type = dumpi
app1.indexing = dumpi
app1.allocation = dumpi
app1.name = parsedumpi
app1.dumpi_metaname = testbgp.meta
```

We have a new parameter `launch_type` set to `dumpi`. This was implicit before, taking the default value of `skeleton`. We also set indexing and allocation parameters to read from the DUMPI trace. The application name is a special app that parses the DUMPI trace. Finally, we direct SST/macro to the DUMPI metafile produced when the trace was collected. To extract the topology information, locate the `.bin` file corresponding to MPI rank 0. To print topology info, run

```
traces> dumpi2ascii -H testbgp-0000.bin
```

which produces the output

```
version=1.1.0
starttime=Fri Nov 22 13:53:58 2013
hostname=R00-M1-N01-J01.challenger
username=<none>
meshdim=3
meshsize=[4, 2, 2]
meshcrd=[0, 0, 0]
```

Here we see that the topology is 3D with extent 4,2,2 in the X,Y,Z directions. At present, the user must still specify the topology in the parameter file. Even though SST/macro can read the topology *dimensions* from the trace file, it cannot read the topology *type*. It could be a torus, dragonfly, or fat tree. The parameter file therefore needs

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

Beyond the topology, the user must also specify the machine model with bandwidth and latency parameters. Again, this is information that cannot be automatically encoded in the trace. It must be determined via small benchmarks like ping-pong. An example file can be found in the test suite in `tests/test_configs/testdumpibgp.ini`.

If no topology info could be recorded in the trace, more work is needed. The only information recorded in the trace is the hostname of each MPI rank. The parameters are almost the same, but with allocation now set to `hostname`. Since no topology info is contained in the trace, a hostname map must be put into a text file that maps a hostname to the topology coordinates. The new parameter file, for a fictional machine called deep thought

```
# Launch parameters
app1.launch_type = dumpi
app1.indexing = dumpi
app1.allocation = hostname
app1.name = parsedumpi
app1.dumpi_metaname = dumpi-2013.09.26.10.55.53.meta
app1.dumpi_mapname = deepthought.map
# Machine parameters
topology_name = torus
topology_geometry = 2 2
```

In this case, we assume a 2D torus with four nodes. Again, 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
app1.indexing = block
app1.allocation = first_available
app1.dumpi_metaname = dumpi-2013.09.26.10.55.53.meta
app1.name = parsedumpi
app1.size = 2
# 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. The trace is only used to generate MPI events and no topology or hostname data is used. The MPI ranks are mapped to physical nodes entirely independent of the trace.

3.9 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 adding a simple filename to the parameter file.

```
call_graph = <fileroot>
```

After running, a `<fileroot>.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 3.15. 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 3.16), 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 3.17). Here we see the function splits execution time between buffering messages (memcpy) and posting headers (Compute Time).

3.10 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
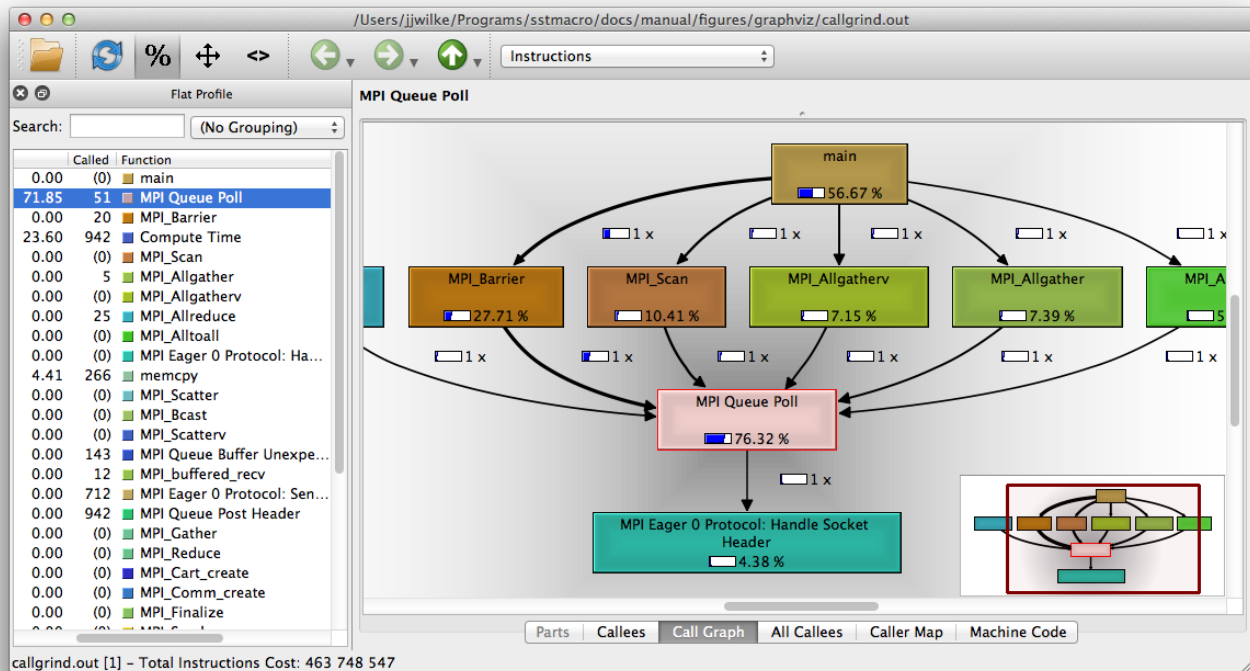
Figure 3.15: QCachegrind GUI

(Figure 3.18). Larger amounts of data (red) are sent to nearest neighbors while decreasing amounts (blue) are sent to MPI ranks further away.

Various spyplots can be activated by boolean parameters in the input file. The most commonly used are the MPI spyplots, for which you must add

```
mpi_spyplot = <fileroot>
```

After running there will be a .csv and .png file in the folder, with e.g. `fileroot = test`

```
example> ls
test.png
test.csv
```

`test.png` shows the number of bytes exchanged between MPI ranks. To extend the analysis you can specify

```
network_spyplot = <fileroot>
```

A new csv/png will appear showing 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.
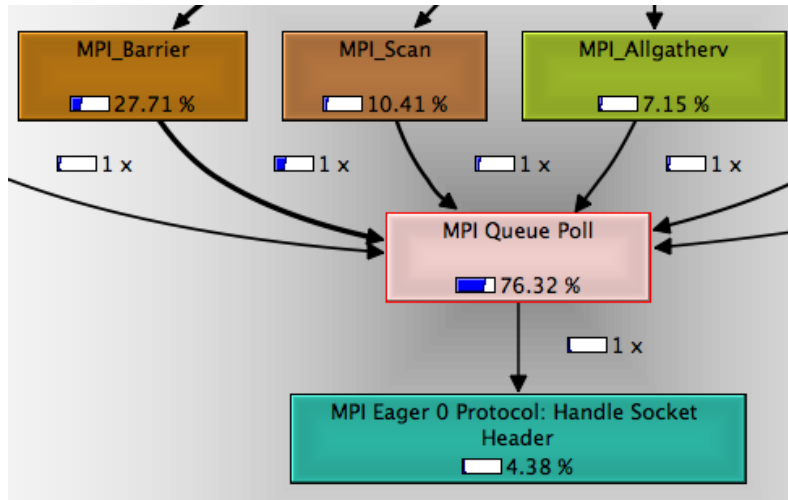
Figure 3.16: QCachegrind Call Graph of MPI Functions
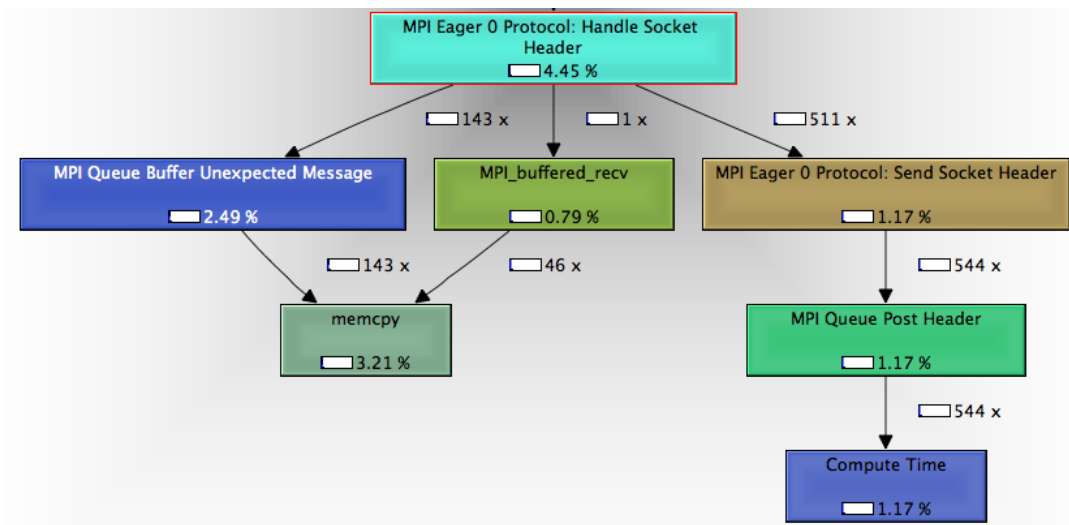
Figure 3.17: QCachegrind Expanded Call Graph of Eager 0 Function

3.11 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 3.19). 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 3.19,
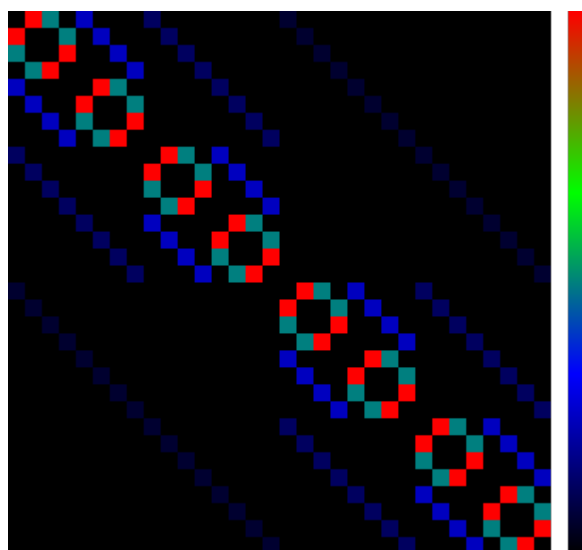
Figure 3.18: Spyplot of Bytes Transferred Between MPI Ranks for MPI_Allreduce
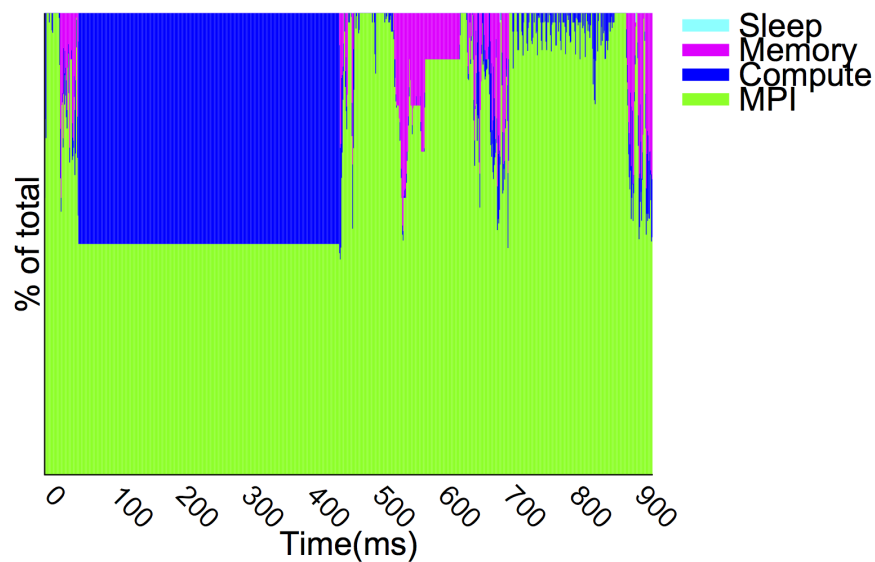
Figure 3.19: Application Activity (Fixed-Time Quanta; FTQ) for Simple MPI Test Suite

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 another input parameter

```
ftq = <fileroot>
```

where the `fileroot` parameter gives a unique prefix for the output files.

After running, two new files appear in the folder: `<fileroot>_app1.p` and `<fileroot>_app1.dat`. `plot_app1.p` is a Gnuplot script that generates the histogram as a postscript file.

```
your_project$ 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.

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.

Using fully namespace parameters, this would be specified as:

```
node.os.ftq.fileroot=<fileroot>
node.os.ftq.epoch=5us
```

3.12 Network Statistics

Here we describe a few of the network statistics that can be collected and the basic mechanism for activating them. These statistics are usually collected on either the NIC, switch crossbar, or switch output buffers.

3.12.1 Message Size Histogram

To active a message size histogram on the NIC to determine the distribution of message sizes, the parameter file should include, for example:

```
nic {
 message_size_histogram {
    fileroot = histogram
    bin_size = 1
    logarithmic = true
 }
}
```

The statistics are activated when the parameter reader sees the namespace `message_size_histogram`. In this case, we ask for a logarithmic distribution. The bin size here is in logarithmic units, i.e. group results in bins corresponding to an exponent range of size 1. An example generated for Nekbone with 1024 processors is in Figure 3.20.
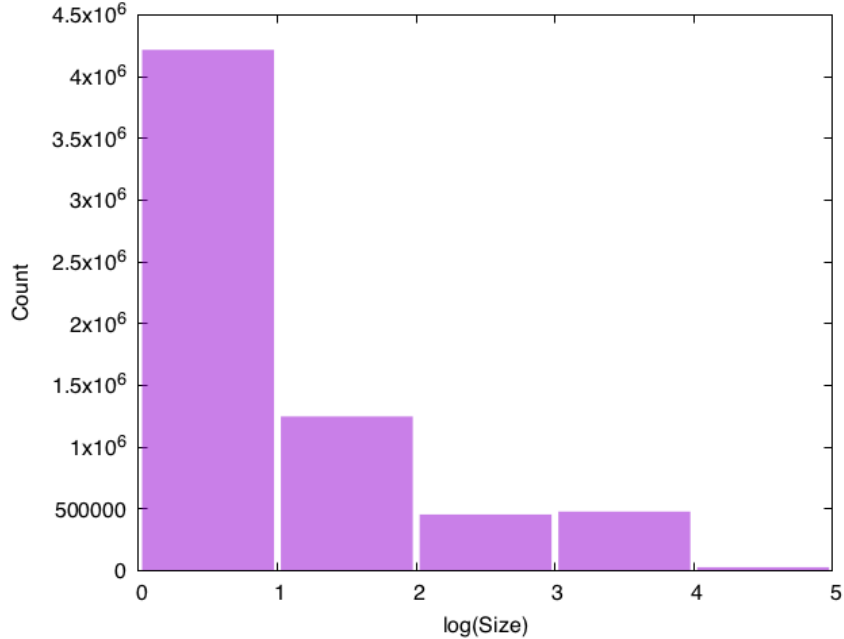
Figure 3.20: Logarithmic histogram of message sizes sent by Nekbone application

3.12.2 Congestion Delay Histogram

A more involved example looks at congestion delays in the application. We want to generate a histogram showing the aggregate delay (relative to zero-congestion baseline) that a packet experiences going from source to destination. By default, packets do not carry fields for measuring congestion. Thus, the packet allocator must be changed. The NIC parameters now become:

```
nic {
 packet_allocator = delay_stats
 ejection {
  stats = delay_histogram
  delay_histogram {
   fileroot = delay
   bin_size = 0.5us
  }
 }
}
```

The delay histogram goes in the ejection namespace since we want to measure packet congestion after it leaves the network. The histogram is not logarithmic and we want to bin files in units of 0.5μs. After running, two files will be generated: a gnuplot script`delay.p` and a corresponding data file `delay.dat`. After running

```
shell>gnuplot delay.p
```

a PNG file `delay.png` is generated. The generated histogram is shown in Figure 3.21. Delays, when occurring in this case, are usually on the order of a few μs.
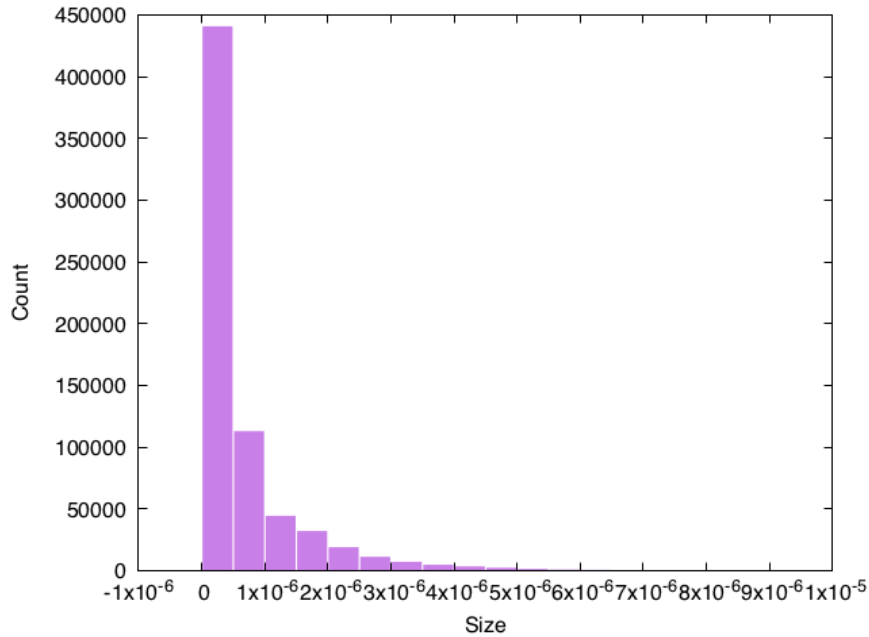
Figure 3.21: Histogram of message sizes sent by Nekbone application

A few more things are actually required to complete the input file. The input file above only tells the NIC to create a histogram. We still have to tell the crossbars and buffers on the network to accumulate the delays on the packet. Otherwise the NIC will just see a zero delay. The following extra parameters are required:

```
switch {
 xbar {
    stats = congestion_delay
 }
 output_buffer {
  stats = congestion_delay
 }
}
```

3.12.3   Congestion Spyplot and Multi-stats

Another way to look for congestion is to create a spyplot. Each row/column is a source/destination pair. In contrast to previous spyplots that show the amount of traffic, a congestion spyplot shows the amount of network congestion experienced sending between two points. Most of the same parameters from the delay histogram are required. However, we now want to collect both a spyplot and the histogram.

```
nic {
 packet_allocator = delay_stats
 ejection {
  stats = multi
  callbacks = congestion_spyplot delay_histogram
  congestion_spyplot {
   fileroot = spyplot
```
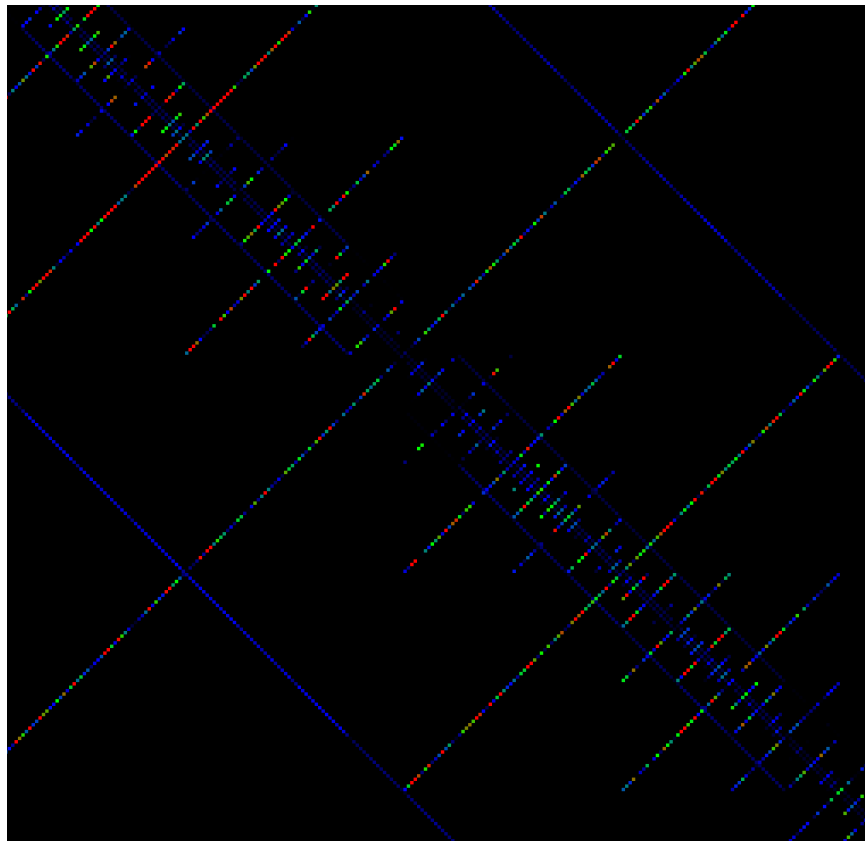
Figure 3.22: Spyplot showing congestion hotspots for certain source/destination pairs for Nekbone.

```
    type = png
    normalization = 100000
  }
  delay_histogram {
    fileroot = delay
    bin_size = 0.5 us
  }
 }
}
```

We set the stats collector to `multi`. We then supply the list of desired stats to the `callbacks` parameter. Zooming into the Nekbone example, we can see congestion hotspots. For the most part, very little congestion appears in Figure 3.22. However (at least for the Torus topology used here), there are a few off-diagonal regions that show some congestion.

Chapter 4

Topologies

The torus topology is straightforward and easy to understand. Here we introduce the basics of other topologies within SST that are more complex and require extra documentation to configure properly. These are generally higher-radix or path-diverse topologies like fat tree, dragonfly, and flattened butterfly. As noted in 3.3, a more thorough and excellent discussions of these topologies is given in "High Performance Datacenter Networks" by Dennis Abts and John Kim.

4.1   Topology Query Utility

Understanding topology inputs and geometries can sometimes be challenging. SST/macro provides an executable for testing topology inputs and doing example coordinate computations. After making and installing, an executable `sstmac_top_info` will appear in the `bin` folder. The invocation of `sstmac_top_info` is exactly the same as the main `sstmac` executable. For the example parameter file named `machine.ini`:

```
topology.name = fattree
topology.geometry = 4 3
```

we run

```
bin> sstmac_top_info -f machine.ini
```

which produces the output

```
Number of nodes:         81
Number of leaf switches: 27
Number of switches:      94
```

detailing the produced geometry. Here the fat tree has a total of 94 switches, 27 of which are "leaf" switches directly connected to compute nodes. The output is followed by the prompt

```
NextInput:
```

One can either enter a single number (switch ID) or set of coordinates. If given a switch ID, the coordinates are computed. If coordinates are given, the switch ID is computed.

```
NextInput: 32
Switch ID maps to coordinates [ 2 0 1 2 ]
NextInput: 2 0 1 2
Coordinates map to switch ID 32
```

The program is just exited with Ctrl-C. The meaning of the above coordinates is detail below for fat tree (Section 4.4).

4.2 Torus

The torus is the simplest topology and fairly easy to understand. We have already discussed basic indexing and allocation as well as routing. More complicated allocation schemes with greater fine-grained control can be used such as the coordinate allocation scheme (see hypercube below for examples) and the node ID allocation scheme (see fat tree below for examples). More complicated Valiant and UGAL routing schemes are shown below for hypercube and dragonfly, but apply equally well to torus.

For torus we illustrate here the Cartesian allocation for generating regular Cartesian subsets. For this, the input file would look like

```
topology.name = torus
topology.geometry = 4 4 4
app1.launch_cmd = aprun −n 8
app1.indexing = block
app1.allocation = cartesian
app1.cart_sizes = 2 2 2
app1.cart_offsets = 0 0 0
```

This allocates a 3D torus of size 4x4x4. Suppose we want to allocate all 8 MPI ranks in a single octant? We can place them all in a 2x2x2 3D sub-torus by specifying the size of the sublock (`cart_sizes`) and which octant (`cart_offsets`). This applies equally well to higher dimensional analogs. This is particularly useful for allocation on Blue Gene machines which always maintain contiguous allocations on a subset of nodes.

This allocation is slightly more complicated if we have multiple nodes per switch. Even though we have a 3D torus, we treat the geometry as a 4D coordinate space with the 4th dimension referring to nodes connected to the same switch, i.e. if two nodes have the 4D coordinates [1 2 3 0] and [1 2 3 1] they are both connected to the same switch. Consider the example below:

```
topology.name = torus
topology.geometry = 4 4 4
topology.concentration = 2
app1.launch_cmd = aprun −n 8
app1.indexing = block
app1.allocation = cartesian
app1.cart_sizes = 2 2 1 2
app1.cart_offsets = 0 0 0 0
```

We allocate a set of switches across an XY plane (2 in X, 2 in Y, 1 in Z for a single plane). The last entry in `cart_sizes` indicates that both nodes on each switch should be used.

4.3 Hypercube

Although never used at scale in a production system, the generalized hypercube is an important topology to understand, particularly for flattened butterfly and dragonfly. The (k,n) generalized hypercube is geometrically an N-dimensional torus

with each dimension having size k (although dimension sizes need not be equal). Here we show a (4,2) generalized hypercube (Figure 4.1). This would be specified in SST as:

```
topology.name = hypercube
topology.geometry = 4 4
```

indicating size 4 in two dimensions.

While a torus only has nearest-neighbor connections, a hypercube has full connectivity within a row and column (Figure 4.1). Any switches in the same row or same column can send packets with only a single hop.
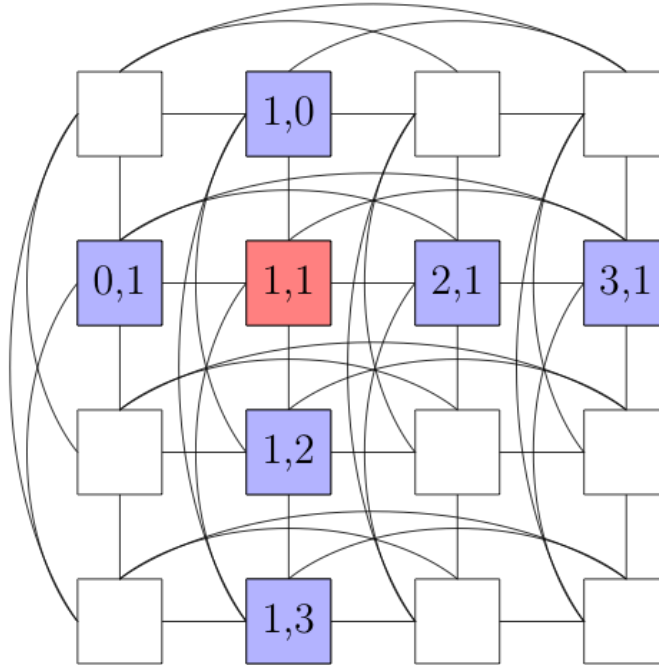
Figure 4.1: Hypercube with links and connections within a row/column

This extra connectivity leads to greater path diversity and higher radix switches. The cost tradeoff is that each link has lower bandwidth than a torus. Whereas a torus has a few fat links connecting switches, a hypercube has many thin links. A hypercube can have more dimensions and be asymmetric, e.g.

```
topology.name = hypercube
topology.geometry = 4 5 6
```

where now we have full connections within horizontal rows, horizontal columns, and vertical columns. Here each switch has radix 12 (3 connections in X, 4 connections in Y, 5 connections in Z).

4.3.1 Allocation and indexing

A hypercube has the same coordinate system as a torus. For example, to create a very specific, irregular allocation on a hyerpcube:

```
app1.launch_cmd = aprun −n 5
app1.indexing = coordinate
app1.allocation = coordinate
app1.coordinate_file = coords.txt
```

and then a coordinate file named `coords.txt`

```
5 2
0 0
0 1
1 1
2 0
3 3
```

The first line indicates 5 entries each with 2 coordinates. Each line then defines where MPI ranks 0-4 will be placed
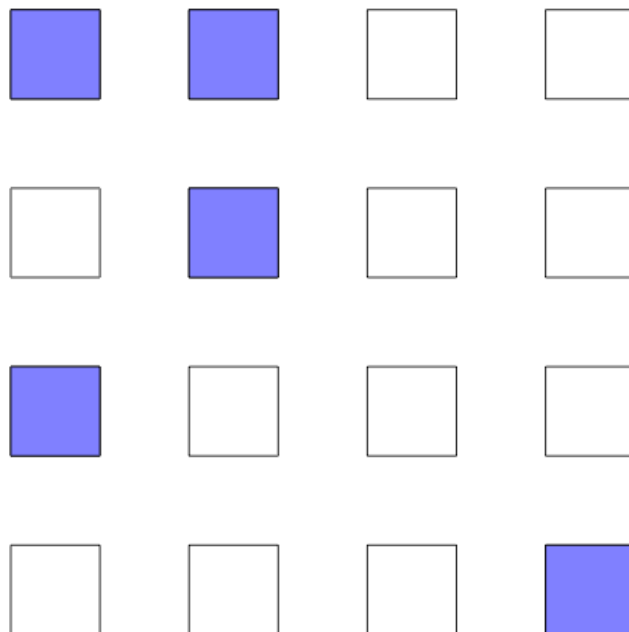
Figure 4.2: Hypercube allocation for given set of coordinates

4.3.2   Routing

Hypercubes allow very path-diverse routing because of its extra connections. In the case of minimal routing (Figure 4.3), two different minimal paths from blue to red are shown. While dimension order routing would rigorously go X then Y, you can still route minimally over two paths either randomly selecting to balance load or routing based on congestion.

To fully maximize path diversity on adversarial traffic patterns, though, path-diverse topologies can benefit from Valiant routing. Here, rather than directly routing to the final destination, packets first route to random intermediate switches on a
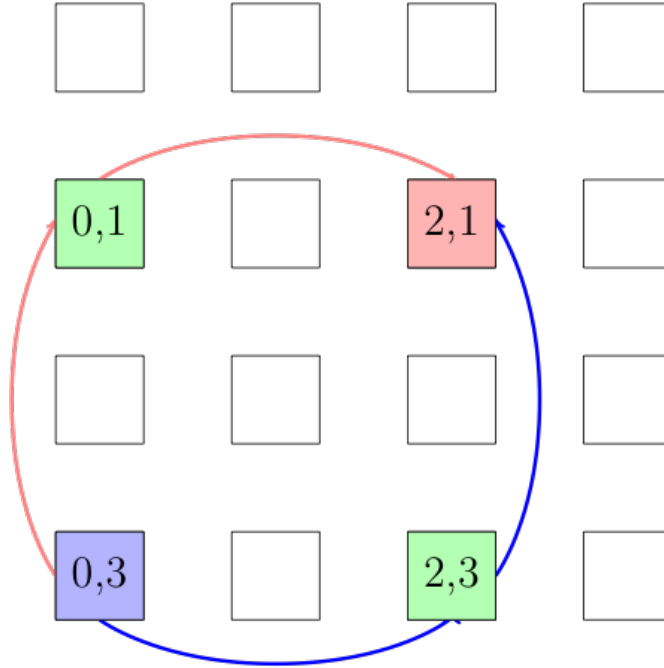
Figure 4.3: Minimal routing within a hypercube showing path diversity. Packet travels from blue to red, passing through green intermediate switches.

minimal path. Then they route again from the intermediate switch to the final destination also on a minimal path (Figure 4.4). Although it increases the hop count and therefore the point-to-point latency, it utilizes more paths and therefore increases the effective point-to-point bandwidth.

4.4 Fat Tree

Within SST, a fat tree is defined by the following parameters:

```
topology.name = fattree
topology.geometry = 4 2
```

The first number, 4, indicates the number of levels in the fat tree. The second number, 2, indicates the radix or branching factor of the tree. The number of compute nodes in this topology is $2^4 = 16$. This is illustrated conceptually in Figure 4.5. The color coding will become clear later. We note this is somewhat confusing since the fat tree appears to have 5 levels. Here the topology is defined by the number of levels containing switches or the number of branches. This is done for a very specific reason. At the final level, you may wish to have a different branching fraction for the compute nodes, e.g.

```
topology.concentration = 1
```
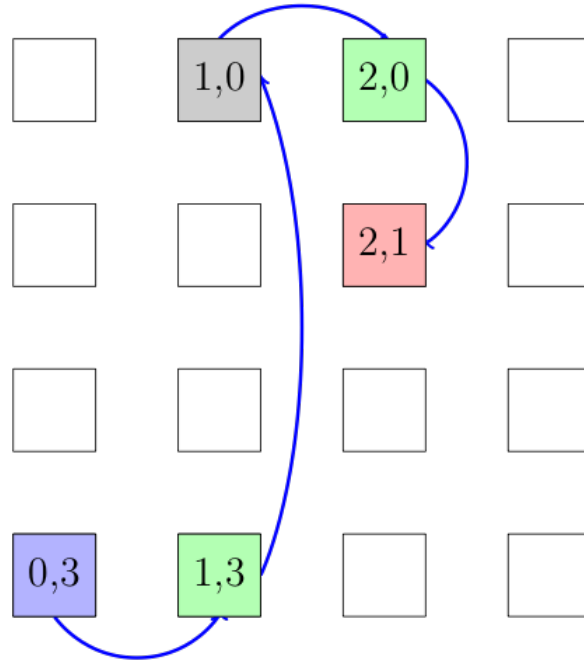
Figure 4.4: Valiant routing within a hypercube. Packet travels from blue to red via a random intermediate destination shown in gray. Additional intermediate switches are shown in green.

This loads the injection bandwidth for the compute node dedicating its own injection switch. If the parameter `network_nodes_per_swi` is omitted, it defaults to the fat tree radix. This case is shown in Figure 4.5 where there are two nodes injecting to the same switch. Higher radix fat trees can be specified, e.g.

```
topology.name = fattree
topology.geometry = 3 4
```

which would have $4^3 = 64$ compute nodes.

In reality, it is not practical to implement a fat tree exactly as shown in Figure 4.5. One would need to buy many non-standard, high capacity switches for the higher levels in the fat-tree. The simple model is available by specifying `simple_fattree` as the topology, and SST will construct special large switches at higher levels. The best practical implementation employs all uniform, commodity switches (Figure 4.6). The fat tree is "virtual" with several commodity switches grouped together to simulate a heavy-weight, high capacity switch at higher levels of the fat tree. The connection between the physical implementation and the conceptual fat tree can easily be seen by the color coding. For example, the second row contains eight switches, but only two virtual switches. Each virtual switch is composed of four commodity switches.

Within SST, each switch is assigned a unique ID, starting from zero in the bottom row and proceeding through the top level. In addition, each compute node is also assigned a unique ID from 0 to 15. The switches can also be defined by a set of coordinates. While the choice of coordinate system for a 3D torus is obvious, the coordinate system for the fat tree is less clear. In SST, we define a 2D mesh coordinate system for the row (level) and column of the switch.
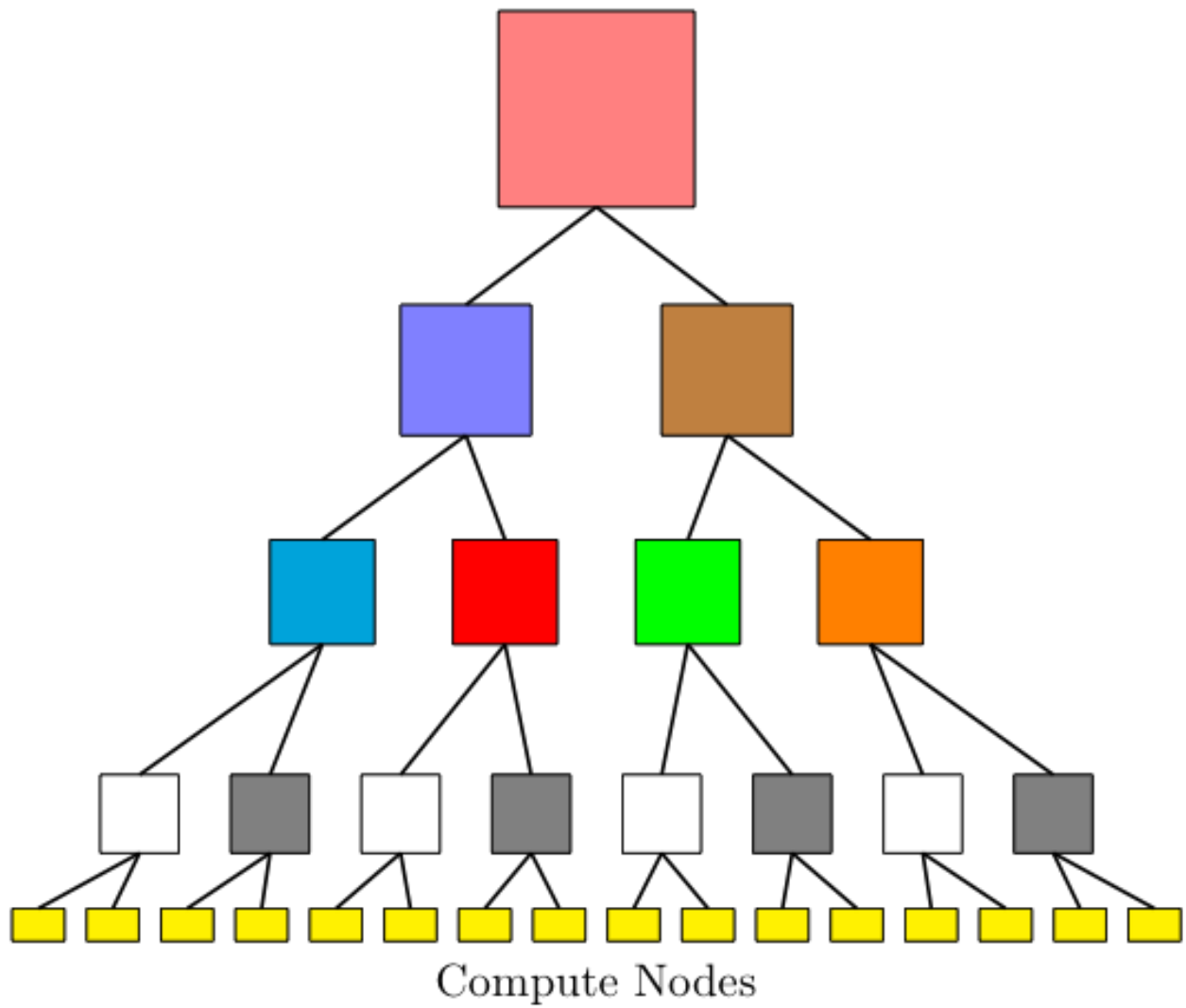
Figure 4.5: Abstract, conceptual picture of Fat Tree topology
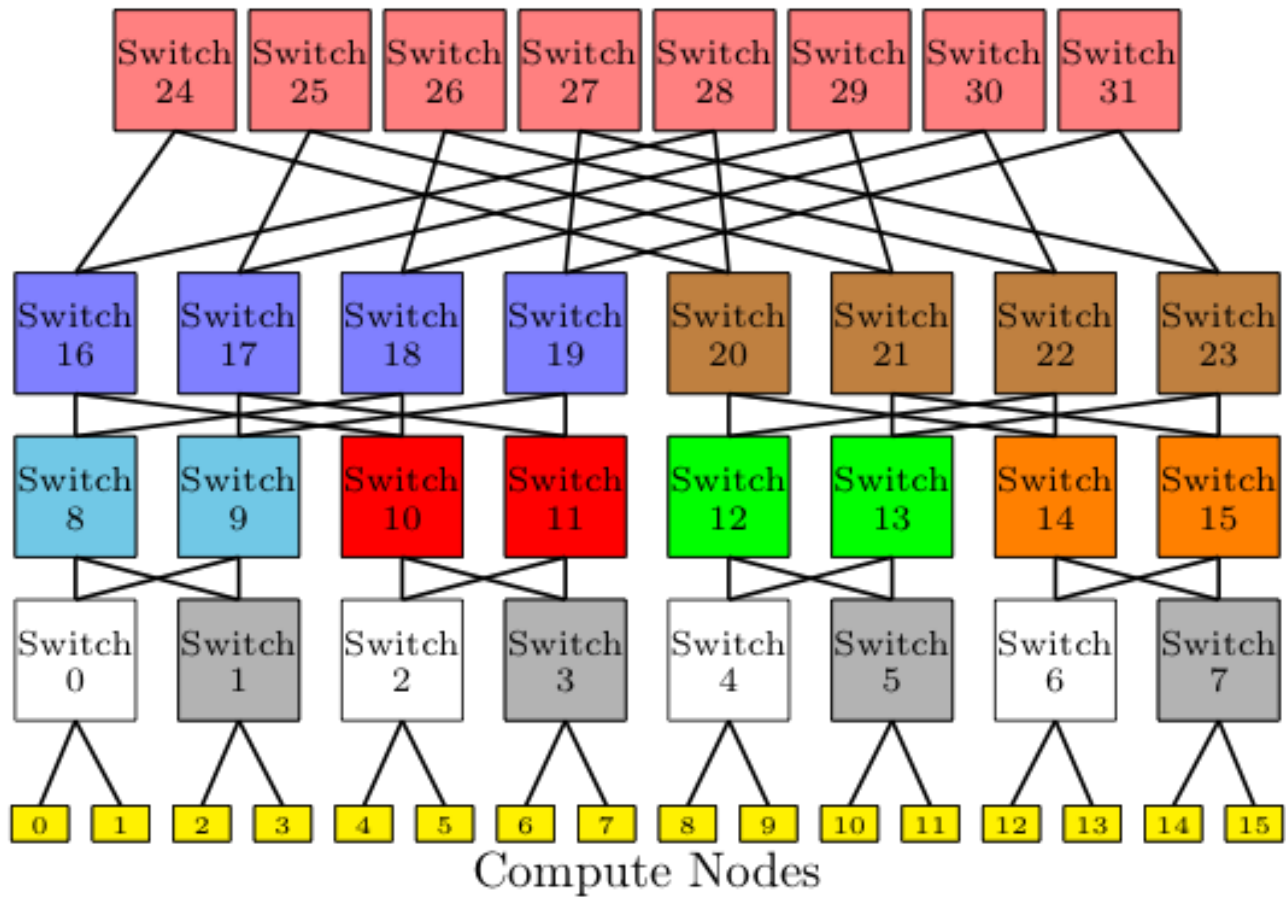
Figure 4.6: Physical implementation of Fat Tree with commodity switches showing ID numbering

4.4.1 Allocation and indexing

The numbering of compute nodes is shown in Figure 4.6. Consider the case

```
app1.launch_cmd = aprun −n 4 −N 1
```

which launches four processes all on distinct nodes. In the simplest allocation and indexing scheme (first available), processes would be placed in order on 0,1,2,3. An alternative allocation/indexing scheme uses the Node ID allocator.

```
app1.allocation = node_id
app1.indexing = node_id
app1.node_id_file = nodes.txt
```

Here `nodes.txt` would contain the number of nodes on the first line, followed by the list of Node IDs, in order, of where to place MPI ranks. For the file

```
4
0
4
8
12
```

Four MPI ranks would be placed in spatially distant parts of the machine.

If indexing differs from allocation (usually because there are multiple MPI ranks per node), both an allocation and an indexing file are needed. Suppose we have:

```
app1.launch_cmd = aprun -n 4 -N 2
```

We then need:

```
app1.allocation = node_id
app1.indexing = node_id
app1.node_id_allocation_file = alloc.txt
app1.node_id_indexing_file = index.txt
```

where the contents of `alloc.txt` are, e.g.

```
2
0
1
```

choosing nodes 0 and 1 in the allocation and then `index.txt` would be, e.g.

```
4
0
1
0
1
```

which round-robin assigns rank 0 to node, rank 1 to node 1, rank 2 to node 0, and so on.

4.4.2 Routing

Fat tree routing is actually straightforward, but can employ path diversity. Suppose you are routing from Node 0 to Node 2 (Figure 4.6). At the first stage, you have no choice. You must route to Switch 1. At the second stage, you can either route to Switch 8 or Switch 9. Suppose you branch to Switch 9. At this point, you are done moving up. The packet now proceeds down the fat-tree. On the downward routing, there is no path diversity. Only a single, minimal route exists to the destination node. In the simplest case, Switch 1 alternates between selecting Switch 8 and Switch 9 to distribute load. In a more complicated scheme, Switch 1 could adaptively route selecting either Switch 8 or Switch 9 based on congestion information.

4.5 Dragonfly

As bandwidth per pin increases, arguments can be made that optimal topologies should be higher radix. A 3D torus is on the low-radix extreme while a hypercube is a high-radix extreme. Unfortunately a hypercube topology is not scalable and the radix quickly becomes too high to efficiently implement. A dragonfly is sometimes viewed as a generalization of flattened
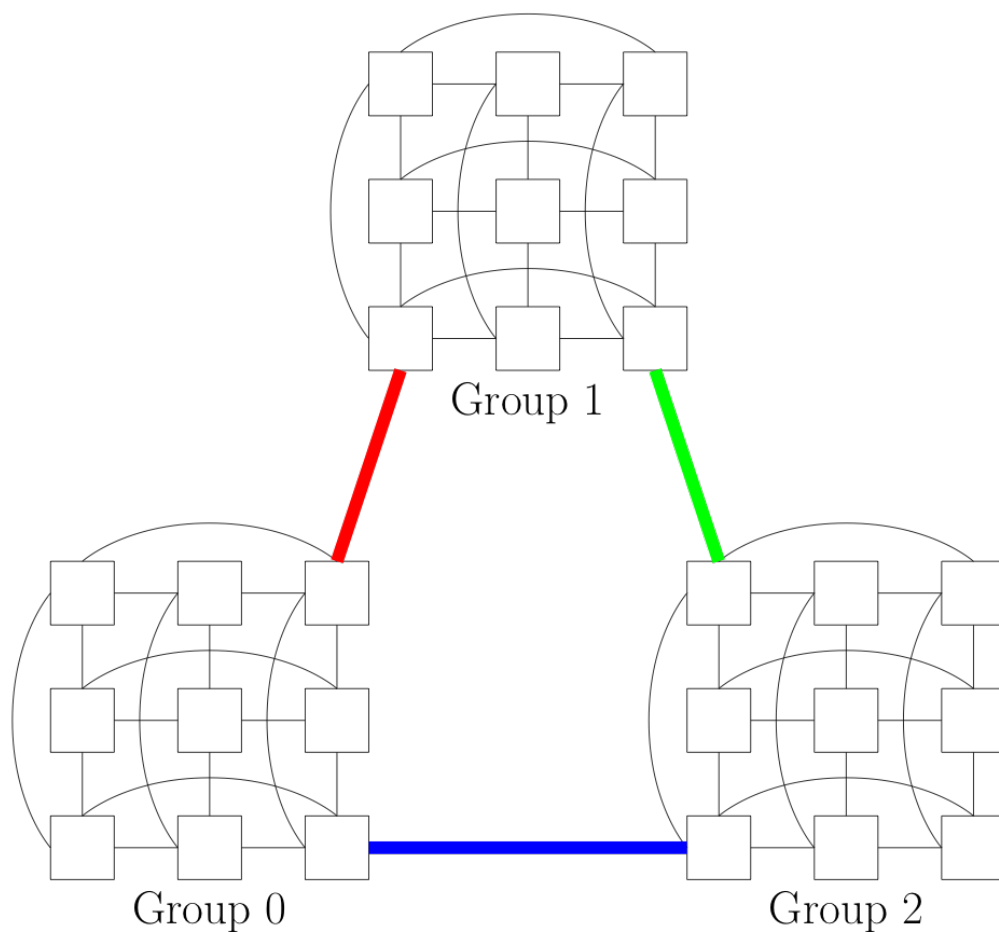
Figure 4.7: Schematic of dragonfly with three groups showing hypercube intragroup links and high bandwidth intergroup global links

butterfly and hypercube topologies with "virtual" switches of very high radix, not dissimilar from the fat-tree implementation with many physical commodity switches composing a single virtual switch. The dragonfly topology (Figure 4.7) is actually quite simple. Small groups are connected as a generalized hypercube with full connectivity within a row or column. Intergroup connections (global links) provide pathways for hopping between groups. A dragonfly is usually understood through three parameters:

- p: number of nodes connected to each router
- a: number of routers in a group
- h: number of global links that each switch has

For simplicity, only three example global links are show for clarity in the picture. For the Cray X630, $a = 96$, $h = 10$, and $p = 4$ so that each router is connected to many other ($h = 10$) groups. The caveat is that in many implementations global

links are grouped together for $h = 2$ or 3 fat global links. These demonstrate well-balanced ratios. In general, scaling out a dragonfly should not increase the size of a group, only the number of groups.

4.5.1 Allocation and indexing

The dragonfly coordinate system is essentially the same as a 3D torus. The group 2D hypercube layout defines X and Y coordinates. The group number defines a Z or G coordinate. Thus the topology in Figure 4.7 would be specified as

```
topology.name = dragonfly
topology.geometry = 3 3 3
```

for groups of size 3×3 with a total of 3 groups. To complete the specification, the number of global links (h) for each router must be given

```
topology.group_connections = 10
```

4.5.2 Routing

It is important to understand the distinction between link bandwidth, channel bandwidth, and pin bandwidth. All topologies have the same pin bandwidth and channel bandwidth (assuming they use the same technology). Each router in a topology is constrained to have the same number of channels (called radix, usually about $k = 64$). The number of channels per link changes dramatically from topology to topology. Low radix topologies like 3D torus can allocate more channels per link, giving higher bandwidth between adjacent routers. Dragonfly is higher radix, having many more connections but having lower bandwidth between adjacent routers. While minimal routing is often sufficient on torus topologies because of the high link bandwidth, dragonfly will exhibit very poor performance with minimal routing. To effectively utilize all the available bandwidth, packets should have a high amount of path diversity. Packets sent between two routers should take as many different paths as possible to maximize the effective bandwidth point-to-point.

Minimal routing itself has a few complications (Figure 4.8). Each router only has a few global links. Thus, traveling from e.g. the blue router at X=3,Y=2,G=0 to the red router at X=1,Y=2,G=2, there is no direct link between the routers. Furthermore, there is no direct link between Groups 0 and 2. Thus packets must route through the purple intermediate nodes. First, the packet hops to X=3,Y=3, G=0. This router has a global link to Group 2, allowing the packet to hop to the next intermediate router at X=1, Y=3, G=2. Finally, the minimal route completes by hopping within Group 2 to the final destination.

To improve on minimal routing, global routing strategies are required (global routing is distinguished here from adaptive routing). Global essentially means "not minimal" and spreads packets along many different paths. The simplest global routing strategy is Valiant routing, which falls in the global, oblivious category (Figure 4.9). Oblivious simply means packets are scattered randomly without measuring congestion. In Valiant routing, each packet does the following:

- Pick a random intermediate node
- Route minimally to random node
- Route minimally from random node to destination node
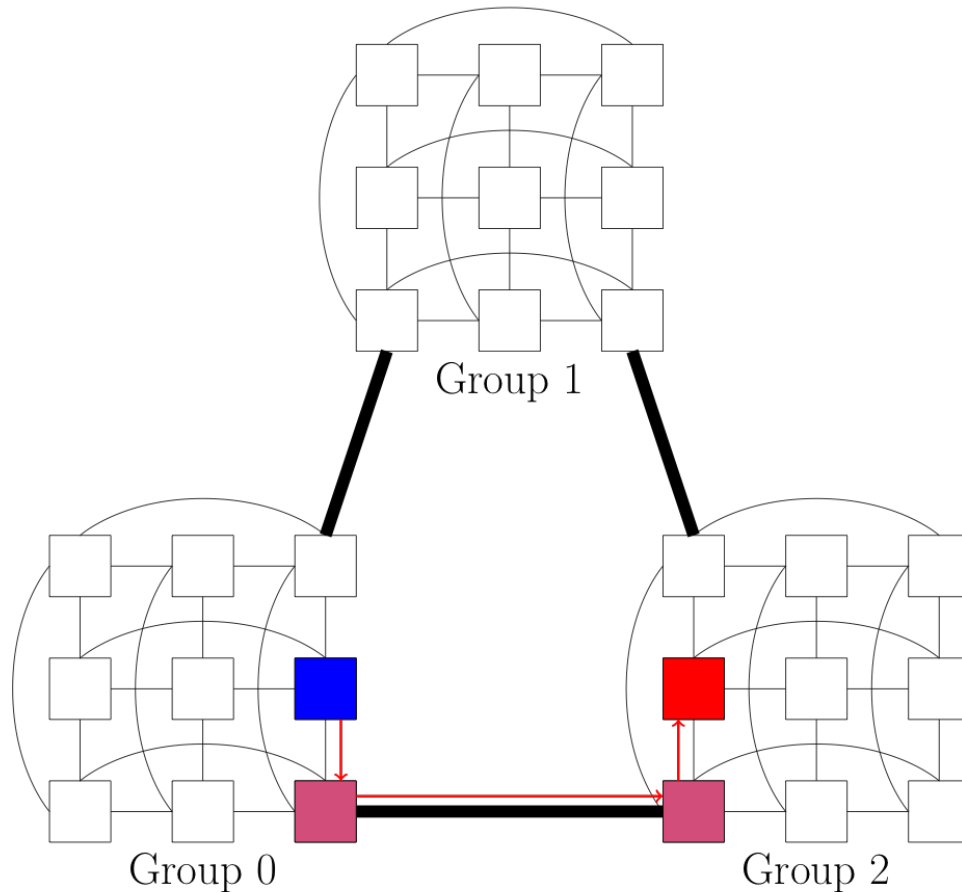
Figure 4.8: Schematic of dragonfly showing minimal route. Traveling between groups requires routing to the correct global link, hopping the global link, then routing within a group to the correct final node.

This is somewhat counterintuitive at first. Rather than go directly to the destination node, packets go out of their way to a random node, shown in Figure 4.9 as the yellow router. Thus, routing from the blue router in Group 0 to the red router in Group 2 first follows the minimal path (green routers) to the randomly selected yellow router in Group 1. From there, a second minimal path is taken through the orange routers to the final destination. In cases with high congestion or even for large messages on a quiet network, this actually improves performance. If a point-to-point message is composed of ten packets, all ten packets will follow different paths to the final destination. This essentially multiplies the maximum bandwidth by a factor of ten. Valiant routing can be specified as

```
router = valiant
```

In contrast, UGAL routing is a global, adaptive strategy, making decisions based on congestion. Because Valiant is oblivious, it often sends too many packets to far away random nodes. Following a Valiant path is only relevant when enough packets fill up router queues, creating congestion. UGAL does the following steps:
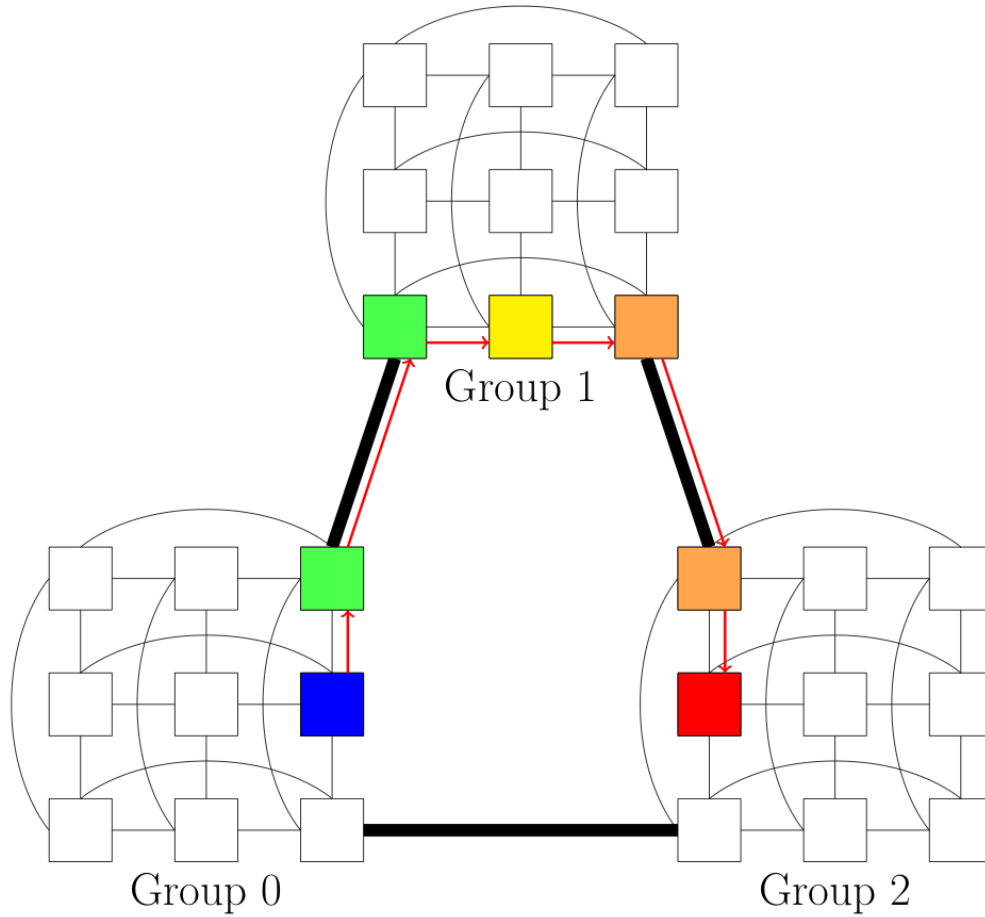
Figure 4.9: Schematic of dragonfly showing Valiant route. Traveling between groups requires routing to a random intermediate node, then routing minimally to the final destination.

- Start routing minimally

- On each step, check congestion (buffer queue depth)

- If congestion is too heavy, switch to Valiant and re-route to random intermediate node. Otherwise stay on minimal path.

UGAL packets stay on a minimal path until congestion forces them to use a Valiant strategy. This routing can be specified as:

```
router = ugal
```

Chapter 5

Applications and Skeletonization

5.1 Basic Application porting

There are three parts to successfully taking a C++ code and turning it into a running application.

- Redirected linkage: Rather than linking to MPI, pThreads, or other parallel libraries (or even calling `hostname`), these functions must be redirected to SST/macro rather than actually executing. You get all redirected linkage for free by using the SST compiler wrappers `sst++` and `sstcc` installed in the `bin` folder.

- Skeletonization: While SST/macro can run in emulation mode, executing your entire application exactly, this is not scalable. To simulate at scale (i.e. 100K or more MPI ranks) you must strip down or "skeletonize" the application to the minimal amount of computation. The energy and time cost of expensive compute kernels are then simulated via models rather than explicitly executed.

- Process encapsulation: Each virtual process being simulated is not an actual physical process. It is instead modeled as a lightweight user-space thread. This means each virtual process has its own stack and register variables, but that is it. Virtual processes share the same address space and the same global variables. Some refactoring may be necessary if you have global variables.

5.2 Redirected linkage

Because of the way libraries are import in the SST core, using external skeleton apps is more complicated than previously. Steps are being taken to re-automate this process, which will hopefully be available in the next subversion. For running the standalone version, this is essentially automatic if using the SST compiler wrappers. The compiler wrappers produce a new executable incorporating your new skeleton.

The wrinkle is external skeletons apps is changing the application's entry point, i.e. `main`. 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 3.5, one needs redirect `main` to a different symbol by defining the macro `sstmac_app_name` at the top of the file containing `main`. This refactoring happens automatically in the SST compiler wrappers.

5.2.1 Loading external skeletons with the integrated core

While the main `libmacro.so` provides the bulk of SST/macro functionality, users may wish to compile and run external skeletons. This gets a bit confusing with SST core since you have an external skeleton for an external element. One can do this manually following the instructions on `http://sst-simulator.org`. You must create an element info struct name `X_eli` for X your library name. You can still use the `sst++` compiler wrappers for building, but you must now manually create a `libX.so` from the compiled object files. The `default.py` script used by `pysstmac` must also be edited. The top lines was previously

```
import sst.macro
```

This only loads the main components, not the external skeleton. You must add

```
import sst.X
```

where X is the name of your skeleton. This causes the core to also load the shared library corresponding to your external skeleton app. If using the SST compiler wrappers, the ELI block and .so file will actually be generated automatically. As shown in Section 2.3, the generate shared library files can be added as the first parameters to the `pysstmac` script.

```
>sst-macro/skeletons/sendrecv> pysstmac librunsstmac.so -f parameters.ini
```

5.3 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.

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 good example of skeletonization can be found in the lulesh2.0.3 example in the skeletons folder.

5.3.1 Basic 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. These functions are all accessible by using the SST compiler wrappers or by adding `#include <sstmac/compute.h>` to your file.

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_memread(long bytes);
void sstmac_memwrite(long bytes);
void sstmac_memcpy(long bytes);
```

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

5.3.2 Detailed compute modeling

The basic compute modeling is not very flexible. In particular, simply computing based on time does not account for congestion delays introduced by things like memory contention. The highly recommended route is a more detailed compute model (but still very simple) that uses the operational intensity (essentially bytes/flops ratio) for a given compute kernel. This informs SST/macro how much stress a given code region puts on either the processor or the memory system. If a kernel has a very high operational intensity, then the kernel is not memory-bound. The means multiple threads can be running the kernel with essentially no memory contention. If a kernel has a very low operational intensity, the kernel is memory bound. A single thread will have good performance, but multiple threads will compete heavily for memory bandwidth. If a kernel has a medium operational intensity, a few concurrent threads may be possible without heavy contention, but as more threads are added the contention will quickly increase.

The function prototype is

```
void
sstmac_compute_detailed(long nflops, long nintops, long bytes);
```

Here `flops` is the number of floating point operations and `bytes` is the number of bytes that hit the memory controller. `bytes` is *not* simply the number of writes/reads that a kernel performs. This is the number of writes/reads that *miss the cache* and hit the memory system. For now, SST/macro assumes a single-level cache and does not distinguish between L1, L2, or L3 cache misses. Future versions may incorporate some estimates of cache hierarchies. However, given the coarse-grained nature

of the simulation, explicit simulation of cache hierarchies is not likely to provide enough improved accuracy or physical insight to justify the increased computational cost. Additional improvements are likely to involve adding parameters for pipelining and prefetching. This is currently the most active area of SST/macro development.

The characterization of a compute kernel must occur outside SST/macro using performance analysis tools like Vtune or PAPI. For the number of flops, it can be quite easy to just count the number of flops by hand. The number of bytes is much harder. For simple kernels like a dot product or certain types of stencil computation, it may be possible to pen-and-paper derive estimates of the number of bytes read/written from memory since every read is essentially a cache miss. In the same way, certain kernels that use small blocks (dense linear algebra), it may be possible to reason *a priori* about the cache behavior. For more complicated kernels, performance metrics might be the only way. Further discussion and analysis of operational intensity and roofline models can be found in "Roofline Model Toolkit: A Practical Tool for Architectural and Program Analysis" by Yung Ju Lo et al. The PDF is available at `http://www.dcs.warwick.ac.uk/~sdh/pmbs14/PMBS14/Workshop_Schedule.html`.

5.3.3 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 or 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 a few options

For applications with heavy dynamic data dependence, we have the following strategies:

- *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.

5.4 Process Encapsulation

As mentioned above, virtual processes are not real, physical processes inside the OS. They are explicitly managed user-space threads with a private stack, but without a private set of global variables. 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. If you have avoided global variables, there is no major issue. If you have read-only global variables with the same value on each machine, there is still no issue. If you have mutable global variables or read-only variables such as `mpi_rank` that differ across processes, there is so minor refactoring that needs to be done.

5.4.1 Manually refactoring global variables

SST/macro provides a complete set of global variable replacements from `#include <sstmac/sstmac_global.h>`, which is automatically included the SST compiler wrappers. Then replace the variable type declaration with the ones that have a `global_` prefix in the header file. To use this file, you 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:

```
1   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.

5.4.2 Automatically refactoring global variables

Tools are currently in use by developers to automatically refactor codes to use no global variables. This involves running the source code through a compiler tool chain that then creates a `struct` encapsulating all global variables into thread-specific classes. This process is only for advanced users and requires developer help.

Chapter 6

Detailed Parameter Listings

The following chapter is organized by parameter namespaces. Tables in each namespace are organized as

Name (type)	Default if not given	Allowed Values	Description

which lists the possible parameter names, allowed values, and brief descriptions. More detailed descriptions of particular parameter values are found in the documentation in previous chapters.

The allowed parameter types are:

int	Any integer
long	Any integer value, but guaranteed not to overflow for long integers
bool	Either "true" or "false" as lowercase string
time	Any valid float value followed by time units (s,ms,us,ns,ps)
freq	Any valid float value followed by frequency units (Hz, MHz, GHz)
bandwidth	Any valid float value followed by bandwidth units (b/s, B/s, Mb/s, MB/s, etc)
byte length	Any positive integer followed by length units (B, KB, MB, GB,TB)
string	An arbitrary string
vector of X	A vector of type X with entries separated by spaces
filepath	A valid filepath to an *existing* file, either absolute or relative
quantiy	A catch-all for a quantity with units. Any of frequency, bandwidth, byte length, or time can be given

6.1   Global namespace

sst_nthread (int)	1	Positive int	Only relevant for multi-threading. Specifying more threads than cores can lead to deadlock.
timestamp_resolution (time)	1ps		Specifies the length of time occupied by 1 timestamp tick - the smallest resolvable time difference. Numerical stability depends on this parameter matching the time scales of the simulation.
serialization_buffer_size (byte length)	512B		Size to allocate for buffers in a pool of serialization buffers. This should set be large enough to handle serialization of most messages, but not so large that significant space is wasted.
cpu_affinity (vector of int)	No default	Invalid cpu IDs give undefined behavior	When in multi-threading, specifies the list of core IDs that threads will be pinned to.

6.2   Namespace "topology"

geometry (vector of int)	No default	See Topology section	Geometry configuration of the topology. For details of the keyword, users should refer to Section 4
name (string)	No default	torus, dragonfly, fat_tree, butterfly, crossbar, tapered_fat_tree	The name of the topology to build. For details, see Section 4
seed (long)	System time		If no value given, random numbers for topology will be generated from system time
concentration (int)	1	Positive int	The number of nodes per network switch. For indirect networks, this is the number of nodes per leaf switch.
num_leaf_switches (int)	No default	Positive int	Only relevant for fat trees. This is the number of switches at the lowest level of the tree that are connected to compute nodes. Depending on how the fat tree is specified, this number may not be required.
k (int)	No default	int $>= 2$	The branching fraction of a fat tree. k=2 is a binary tree. k=4 is a quad-tree.
l (int)	No default	Positive int	The number of levels in a fat tree.
num_inj_switches_per_subtree	No default	Positive int	For a tapered tree, the number of injection switches, N_{inj}, within an aggregation tree that connect directly toc ompute nodes.
num_agg_switches_per_subtree	No default	Positive int	For a tapered tree, the number of aggregations witches per aggregation tree linking injection switches to the core.
num_agg_subtrees	No default	Positive int	For a tapered fat tree with 3 levels (injection, aggregation, core), this gives the number, N_{agg}, of aggregation subtrees. To find the total number, N_{tot} of injection (leaf) switches, we have $N_{tot} = N_{agg} \times N_{inj}$.
num_core_switches	No default	Positive int	The total number of core switches in a tapered tree linking the individual aggregation trees.
group_connections (int)	No default	Positive int	For dragonfly, the number of intergroup connections on each switch in a Dragonfly group
redundant (vector of int)	vector of 1's	Positive ints	For Cartesian topologies (hypercube, dragonfly, torus) this specifies a bandwidth (redundancy) multiplier for network links in each dimension.

6.3   Namespace "node"

model (string)	simple	simple	The type of node model (level of detail) for node-level operations
services (vector of strings)	Empty	Valid service names	For details, see section on distributed services in developer's manual. Advanced feature.

6.3.1   Namespace "node.nic"

model (string)	No default	pisces, logP	The type of NIC model (level of detail) for modeling injection of messages (flows) to/from the network.
packetizer (string)	cut_through	merlin, simple, cut_through	The type of packetizer for injecting flows into the network. Merlin is part of sst-elements. Simple and cut-through use PISCES
packet_allocator (string)	pisces	pisces	The packet object used by the NIC. Certain type of statistics or congestion models may require custom packets with extra fields.
negligible_size (byte length)	256B		Messages (flows) smaller than size will not go through detailed congestion modeling. They will go through a simple analytic model to compute the delay.

Namespace "node.nic.delay_histogram"

fileroot (string)	No default		The path of a file to be created (or overwritten) where results from the statistics will be placed.
bin_size (quantity)	No default	Positive value	For the quantity being collected (bytes,time,etc), this determines how large a value range should be binned together.
num_bins (int)	20	Positive int	Determines the number of initial bins to allocate. If number is too small, simulation will allocate more. This is just a guess for optimization.
logarithmic (bool)	false		Whether to use a logarithmic axis for binning.

Namespace "node.nic.congestion_spyplot"

fileroot (string)	No default		The path of a file to be created (or overwritten) where results from the statistics will be placed.
type (string)	ascii	ascii, png	Whether to generate a simple traffic matrix in an ASCII file or to generate a PNG image of the traffic pattern
normalization (long)	Ignored	Positive int	Determines a normalization value that will effect the color scales of an output PNG file. By default, if not specified, the spyplot will determine the largest value and normalize all values for the output PNG to be 0.0-1.0. Values larger than 1.0 will have the same color as 1.0. Useful for having two PNG files with different values have the same color scales.

Namespace "node.nic.traffic_matrix"

fileroot (string)	No default		The path of a file to be created (or overwritten) where results from the statistics will be placed.
type (string)	ascii	ascii, png	Whether to generate a simple traffic matrix in an ASCII file or to generate a PNG image of the traffic pattern
normalization (long)	Ignored	Positive int	Determines a normalization value that will effect the color scales of an output PNG file. By default, if not specified, the spyplot will determine the largest value and normalize all values for the output PNG to be 0.0-1.0. Values larger than 1.0 will have the same color as 1.0. Useful for having two PNG files with different values have the same color scales.

Namespace "node.nic.local_bytes_sent"

fileroot (string)	No default		The path of a file to be created (or overwritten) where results from the statistics will be placed.

Namespace "node.nic.global_bytes_sent"

fileroot (string)	No default		The path of a file to be created (or overwritten) where results from the statistics will be placed.

Namespace "node.nic.message_size_histogram"

fileroot (string)	No default		The path of a file to be created (or overwritten) where results from the statistics will be placed.
bin_size (quantity)	No default	Positive value	For the quantity being collected (bytes,time,etc), this determines how large a value range should be binned together.
num_bins (int)	20	Positive int	Determines the number of initial bins to allocate. If number is too small, simulation will allocate more. This is just a guess for optimization.
logarithmic (bool)	false		Whether to use a logarithmic axis for binning.

Namespace "node.nic.ejection"

These parameters do not need to be specified, but can be given. Generally, the simulation assumes an infinite buffer size (unlimited memory) and no latency. All other parameters can be filled in from `node.nic.injection`.

stats (string)	null	delay_histogram, congestion_spyplot, multi, null	The type of statistics to collect from packets leaving or arriving at the NIC. Null indicates no statistics collected. For details on the other types of statistics, see Section 3.12
callbacks (vector of string)	No default	Any valid stats name	If stats is set to multi,then multiple different stats can be given here
arbitrator (string)	cut_through	null, simple, cut_through	Bandwidth arbitrator for PISCES congestion modeling. Null uses simple delays with no congestion. Simple uses store-and-forward that is cheap to compute, but can have severe latency errors for large packets. Cut-through approximates pipelining of flits across stages.
latency (time)	No default		If given, overwrites the send and credit latency parameters. Depending on component, the entire latency may be put on either the credits or the send.
bandwidth	No default		The bandwidth of the arbitrator
send_latency (time)	No default		The latency to send a packet to the next stage in the network. This can be omitted if the generic latency parameter is given (see above).
credit_latency (time)	No default		The latency to send a credit to the previous network stage. This can be omitted if the generic latency parameter is given (see above).
num_vc (int)	Computed	Positive int	If not specified, SST will estimate the number of virtual channels based on the topology and routing. If given and parameter is too small, the system can deadlock. If too large, buffer resources will be underutilized.
credits (byte length)	No default		The number of initial credits for the component. Corresponds to an input buffer on another component. In many cases, SST/macro can compute this from other parameters and fill in the value. In some cases, it will be required.
mtu (byte length)	1024B		The packet size. All messages (flows) will be broken into units of this size.

Namespace "node.nic.injection"

stats (string)	null	delay_histogram, null	The type of statistics to collect from packets leaving or arriving at the NIC. Null indicates no statistics collected. For details on the other types of statistics, see Section 3.12
arbitrator (string)	cut_through	null, simple, cut_through	Bandwidth arbitrator for PISCES congestion modeling. Null uses simple delays with no congestion. Simple uses store-and-forward that is cheap to compute, but can have severe latency errors for large packets. Cut-through approximates pipelining of flits across stages.
latency (time)	No default		If given, overwrites the send and credit latency parameters. Depending on component, the entire latency may be put on either the credits or the send.
bandwidth	No default		The bandwidth of the arbitrator
send_latency (time)	No default		The latency to send a packet to the next stage in the network. This can be omitted if the generic latency parameter is given (see above).
credit_latency (time)	No default		The latency to send a credit to the previous network stage. This can be omitted if the generic latency parameter is given (see above).
num_vc (int)	Computed	Positive int	If not specified, SST will estimate the number of virtual channels based on the topology and routing. If given and parameter is too small, the system can deadlock. If too large, buffer resources will be underutilized.
credits (byte length)	No default		The number of initial credits for the component. Corresponds to an input buffer on another component. In many cases, SST/macro can compute this from other parameters and fill in the value. In some cases, it will be required.
mtu (byte length)	1024B		The packet size. All messages (flows) will be broken into units of this size.

6.3.2 Namespace "node.memory"

model (string)	No default	logP, pisces	The type of memory model (level of detail) for modeling memory transactions.
arbitrator (string)	cut_through	null, simple, cut_through	The type of arbitrator. See arbitrator descriptions above.
latency (time)	No default		The latency of single memory operation
total_bandwidth	No default		The total memory bandwidth possible across all memory controllers.
max_single_bandwidth	Computed		The maximum memory bandwidth of a single stream of requests. Essentially the bandwidth of a single memory controller. If not given, this defaults the value of total_bandwidth.

6.3.3 Namespace "node.os"

compute_scheduler (string)	simple	simple, cpuset	The level of detail for scheduling compute tasks to cores. Simple looks for any empty core. cpuset allows bitmasks to be set for defining core affinities.
stack_size (byte length)	32KB		The size of user-space thread stack to allocate for each virtual application
stack_protect (bool)	false		Whether to put special mprotect regions around the thread stacks to turn stack overflow errors into seg faults
stack_chunk_size (byte length)	1MB		The size of memory to allocate at a time when allocating new thread stacks. Rather than allocating one thread stack at a time, multiple stacks are allocated and added to a pool as needed.

Namespace "node.os.call_graph"

fileroot (string)	No default		The path of a file to be created (or overwritten) where results from the statistics will be placed.

Namespace "node.os.ftq"

fileroot (string)	No default		The path of a file to be created (or overwritten) where results from the statistics will be placed.
epoch (time)	No default		The length of simulation time to treat as a single interval. This is much like a histogram bin, except the bin is partitioned into multiple categories. If too small, not enough data will be collected per interval to give reasonable looking results. If too large, significant changes in system activity over time will not be resolved.

6.3.4   Namespace "node.proc"

ncores (int)	No default	Positive int	The number of cores contained in a processor (socket). Total number of cores for a node is $ncores \times nsockets$.
frequency	No default		The baseline frequency of the node
parallelism (double)	1.0	Positive number	Fudge factor to account for superscalar processor. Number of flops per cycle performed by processor.

6.4   Namespace "netlink"

model (string)	null	null, pisces	The type of netlink congestion model to use. If null, no netlink blocks will be used.
concentration (int)	No default	Positive int	The number of nodes connected to a given netlink. The node concentration per switch must be greater than or equal to the node concentration per netlink.
num_tiles (int)	1	Positive int	The number of router tiles the netlink connects to. Only relevant for tiled router models. Advanced usage.

6.4.1 Namespace "netlink.injection"

arbitrator (string)	cut_through	null, simple, cut_through	Bandwidth arbitrator for PISCES congestion modeling. Null uses simple delays with no congestion. Simple uses store-and-forward that is cheap to compute, but can have severe latency errors for large packets. Cut-through approximates pipelining of flits across stages.
latency (time)	No default		If given, overwrites the send and credit latency parameters. Depending on component, the entire latency may be put on either the credits or the send.
bandwidth	No default		The bandwidth of the arbitrator
send_latency (time)	No default		The latency to send a packet to the next stage in the network. This can be omitted if the generic latency parameter is given (see above).
credit_latency (time)	No default		The latency to send a credit to the previous network stage. This can be omitted if the generic latency parameter is given (see above).
num_vc (int)	Computed	Positive int	If not specified, SST will estimate the number of virtual channels based on the topology and routing. If given and parameter is too small, the system can deadlock. If too large, buffer resources will be underutilized.
credits (byte length)	No default		The number of initial credits for the component. Corresponds to an input buffer on another component. In many cases, SST/macro can compute this from other parameters and fill in the value. In some cases, it will be required.
mtu (byte length)	1024B		The packet size. All messages (flows) will be broken into units of this size.

6.4.2 Namespace "netlink.ejection"

arbitrator (string)	cut_through	null, simple, cut_through	Bandwidth arbitrator for PISCES congestion modeling. Null uses simple delays with no congestion. Simple uses store-and-forward that is cheap to compute, but can have severe latency errors for large packets. Cut-through approximates pipelining of flits across stages.
latency (time)	No default		If given, overwrites the send and credit latency parameters. Depending on component, the entire latency may be put on either the credits or the send.
bandwidth	No default		The bandwidth of the arbitrator
send_latency (time)	No default		The latency to send a packet to the next stage in the network. This can be omitted if the generic latency parameter is given (see above).
credit_latency (time)	No default		The latency to send a credit to the previous network stage. This can be omitted if the generic latency parameter is given (see above).
num_vc (int)	Computed	Positive int	If not specified, SST will estimate the number of virtual channels based on the topology and routing. If given and parameter is too small, the system can deadlock. If too large, buffer resources will be underutilized.
credits (byte length)	No default		The number of initial credits for the component. Corresponds to an input buffer on another component. In many cases, SST/macro can compute this from other parameters and fill in the value. In some cases, it will be required.
mtu (byte length)	1024B		The packet size. All messages (flows) will be broken into units of this size.

6.5 Namespace "mpi"

test_delay (time)	0		The minimum time spent by MPI on each MPI_Test call
iprobe_delay (time)	0		The minimum time spent by MPI on each MPI_Iprobe call

6.5.1 Namespace "mpi.queue"

max_vshort_msg_size (byte length)	512B		The maximum size to use the very short message protocol. Small messages are sent eagerly using special pre-allocated mailbox buffers. Sends complete immediately.
max_eager_msg_size (byte length)	8KB		The maximum size to use the RDMA eager protocol. This also uses buffers to send message, but instead of using pre-allocated mailboxes, it coordinates an RDMA get. Sends complete immediately.
post_rdma_delay (time)	0		The minimum time spent by MPI posting each RDMA operation
post_header_delay (time)	0		The mimimum time spent by MPI sending headers into pre-allocated mailboxes
poll_delay (time)	0		The minimum time spent by MPI each time it polls for incoming messages

6.6 Namespace "switch"

model (string)	No default	logP, pisces	The type of switch model (level of detail) for modeling network traffic.
buffer_size (byte length)	No default		The size of input and output buffers on each switch. This determines the number of credits available to other components

6.6.1 Namespace "switch.router"

name (string)	No default	minimal, valiant, ugal, fat_tree	The name of the routing algorithm to use for routing packets.
ugal_threshold (int)	0		The minimum number of network hops required before UGAL is considered. All path lengths less than value automatically use minimal.

6.6.2 Namespace "switch.output_buffer"

stats (string)	null	delay_histogram, byte_hops, null	The type of statistics to collect from packets leaving or arriving at the switch output buffers. Null indicates no statistics collected. For details on the other types of statistics, see Section 3.12

Namespace "switch.output_buffer.delay_histogram"

fileroot (string)	No default		The path of a file to be created (or overwritten) where results from the statistics will be placed.
bin_size (quantity)	No default	Positive value	For the quantity being collected (bytes,time,etc), this determines how large a value range should be binned together.
num_bins (int)	20	Positive int	Determines the number of initial bins to allocate. If number is too small, simulation will allocate more. This is just a guess for optimization.
logarithmic (bool)	false		Whether to use a logarithmic axis for binning.

Namespace "switch.output_buffer.byte_hops"

fileroot (string)	No default		The path of a file to be created (or overwritten) where results from the statistics will be placed.

6.6.3   Namespace "switch.xbar"

stats (string)	null	bytes_sent, null	The type of statistics to collect from packets leaving or arriving at the switch crossbar. Null indicates no statistics collected. For details on the other types of statistics, see Section 3.12
arbitrator (string)	cut_through	null, simple, cut_through	Bandwidth arbitrator for PISCES congestion modeling. Null uses simple delays with no congestion. Simple uses store-and-forward that is cheap to compute, but can have severe latency errors for large packets. Cut-through approximates pipelining of flits across stages.
latency (time)	No default		If given, overwrites the send and credit latency parameters. Depending on component, the entire latency may be put on either the credits or the send.
bandwidth	No default		The bandwidth of the arbitrator
send_latency (time)	No default		The latency to send a packet to the next stage in the network. This can be omitted if the generic latency parameter is given (see above).
credit_latency (time)	No default		The latency to send a credit to the previous network stage. This can be omitted if the generic latency parameter is given (see above).
num_vc (int)	Computed	Positive int	If not specified, SST will estimate the number of virtual channels based on the topology and routing. If given and parameter is too small, the system can deadlock. If too large, buffer resources will be underutilized.
credits (byte length)	No default		The number of initial credits for the component. Corresponds to an input buffer on another component. In many cases, SST/macro can compute this from other parameters and fill in the value. In some cases, it will be required.
mtu (byte length)	1024B		The packet size. All messages (flows) will be broken into units of this size.

Namespace "switch.xbar.bytes_sent"

fileroot (string)	No default		The path of a file to be created (or overwritten) where results from the statistics will be placed.

6.6.4   Namespace "switch.link"

arbitrator (string)	cut_through	null, simple, cut_through	Bandwidth arbitrator for PISCES congestion modeling. Null uses simple delays with no congestion. Simple uses store-and-forward that is cheap to compute, but can have severe latency errors for large packets. Cut-through approximates pipelining of flits across stages.
latency (time)	No default		If given, overwrites the send and credit latency parameters. Depending on component, the entire latency may be put on either the credits or the send.
bandwidth	No default		The bandwidth of the arbitrator
send_latency (time)	No default		The latency to send a packet to the next stage in the network. This can be omitted if the generic latency parameter is given (see above).
credit_latency (time)	No default		The latency to send a credit to the previous network stage. This can be omitted if the generic latency parameter is given (see above).
num_vc (int)	Computed	Positive int	If not specified, SST will estimate the number of virtual channels based on the topology and routing. If given and parameter is too small, the system can deadlock. If too large, buffer resources will be underutilized.
credits (byte length)	No default		The number of initial credits for the component. Corresponds to an input buffer on another component. In many cases, SST/macro can compute this from other parameters and fill in the value. In some cases, it will be required.
mtu (byte length)	1024B		The packet size. All messages (flows) will be broken into units of this size.

6.6.5   Namespace "switch.ejection"

This namespace is not actually required. If unspecified, all of the values here will be filled in from `nic.injection`.

arbitrator (string)	cut_through	null, simple, cut_through	Bandwidth arbitrator for PISCES congestion modeling. Null uses simple delays with no congestion. Simple uses store-and-forward that is cheap to compute, but can have severe latency errors for large packets. Cut-through approximates pipelining of flits across stages.
latency (time)	No default		If given, overwrites the send and credit latency parameters. Depending on component, the entire latency may be put on either the credits or the send.
bandwidth	No default		The bandwidth of the arbitrator
send_latency (time)	No default		The latency to send a packet to the next stage in the network. This can be omitted if the generic latency parameter is given (see above).
credit_latency (time)	No default		The latency to send a credit to the previous network stage. This can be omitted if the generic latency parameter is given (see above).
num_vc (int)	Computed	Positive int	If not specified, SST will estimate the number of virtual channels based on the topology and routing. If given and parameter is too small, the system can deadlock. If too large, buffer resources will be underutilized.
credits (byte length)	No default		The number of initial credits for the component. Corresponds to an input buffer on another component. In many cases, SST/macro can compute this from other parameters and fill in the value. In some cases, it will be required.
mtu (byte length)	1024B		The packet size. All messages (flows) will be broken into units of this size.

6.7 Namespace "appN"

This is a series of namespaces `app1`, `app2`, and so on for each of the launched applications.

name (string)	No default	parsedumpi, cxx_full_main, cxx_empty_main	The name of the application to launch. Very few applications are built-in. Registration of external apps is shown starting in Section 3.5.
size (int)	No default	Positive int	The number of procs (MPI ranks) to launch. If launch_cmd given, this parameter is not required.
start (int)	0		The time at which a launch request for the application will be made
concentration (int)	1	Positive int	The number of procs (MPI ranks) per compute node
core_affinities (vector of int)	Empty		
launch_cmd (string)	No default	Valid aprun or srun	This uses a launch command as would be found with ALPS or SLURM launchers on real systems, e.g. aprun -n 4 -N 1
indexing (string)	block	block, random, cart, node_id, coordinate	The indexing scheme for assign proc ID (MPI rank number) to compute nodes
node_id_mapper_file (filepath)	No default		If using Node ID indexing, the file containing the node ID index list
random_indexer_seed (long)	System time		The seed to use for a random allocation. If not specified, system time is used.
allocation (string)	first_available	first_available, random, cart, node_id, coordinate	The scheme to use for allocating compute nodes to a given job.
random_allocation_seed (long)	System time		For random allocation policy. If unspecified, system time is used as the seed.
node_id_allocation_file (filepath)	No default		If using Node ID allocation, the file containing the list of node IDs to allocate for the job
dumpi_metaname (filepath)	No default		If running DUMPI trace, the location of the metafile for configuring trace replay
coordinate_file (filepath)	No default		If running using coordinate allocation or indexing, the path to the file containing the node coordinates of each proc (MPI rank)
cart_sizes (vector of int)	No default		Launch a contiguous block of nodes in a Cartesian topology. This gives the size of each dimension in the block.
cart_offsets (vector of int)	No default		Launch a contiguous block nodes in a Cartesian topology. This gives the offset in each dimension where the block begins.
parsedumpi_timescale (double)	1.0	Positive float	If running DUMPI traces, scale compute times by the given value. Values less than 1.0 speed up computation. Values greater than 1.0 slow down computation.
parsedumpi_terminate_percent (int)	100	1-100	Percent of trace. Can be used to terminate large traces early