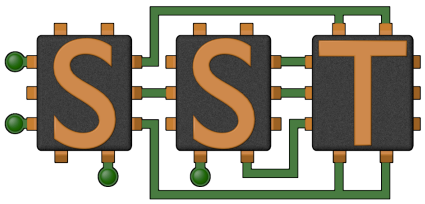
SST/macro 9.0: User's Manual

Sandia National Labs
Livermore, CA

May 22, 2019



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, but it is limited.

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++ 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. 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. Skeletonization and UQ are the two main elements in the "canonical" SST/macro workflow (Figure 1.1).

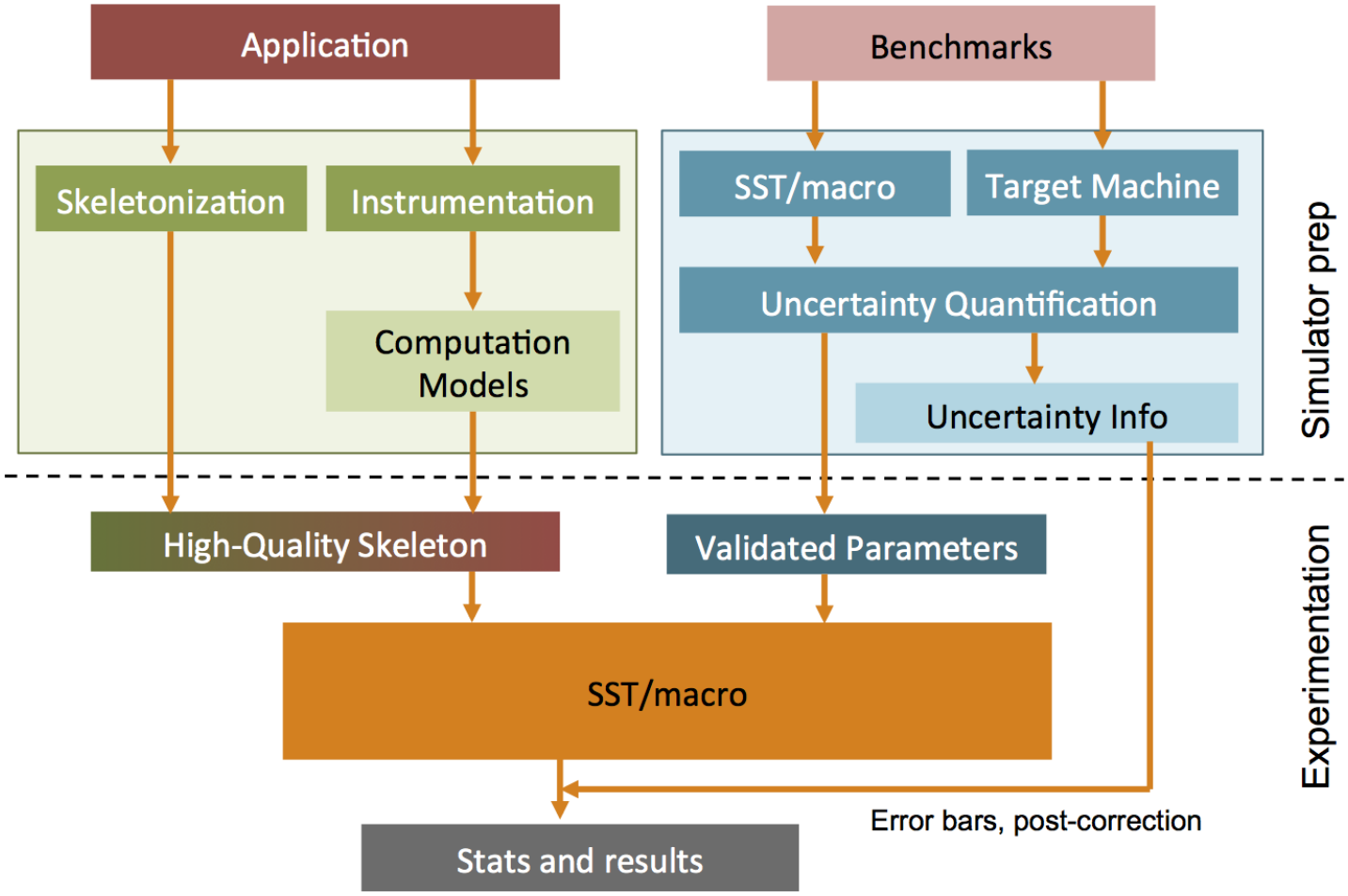


Figure 1.1: SST/macro workflow.

Because of its popularity, MPI is one of our main priorities in providing programming model support. Some MPI-3 functions and MPI one-sided functions are not implemented. This will lead to compile errors with an obvious "not implement" compiler message.

1.2 Preview of Things to Come

Suppose you have the basic MPI application below that executes a simple send/recv operation. One could use `mpicc` or `mpic++` to compile and run as an actual MPI program. This requires spawning all the processes and running them in parallel. Suppose, however, you wanted to simulate an entire MPI job launch within a single simulator process. This might prove very useful for debugging since you could just run GDB or Valgrind on a single process. It might take a while, but for small runs (16 ranks or so) you could debug right on your laptop just as you do for a serial program.

```
int size = atoi(argv[1]);
if (rank == 0){
 int partner = 1;
  MPI_Send(buffer, size, MPI_INT, partner, tag, MPI_COMM_WORLD);
} else {
  int partner = 0;
  MPI_Recv(buffer, size, MPI_INT, partner, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
MPI_Barrier(MPI_COMM_WORLD);

if (rank == 0){
  printf("Rank 0 finished at t=%8.4f ms\n", MPI_Wtime()*1e3);
}
```

This is exactly the functionality that SST/macro provides. Instead of using `mpic++`, you compile the code with `sst++`. This modifies your code and intercepts MPI calls, running them through the simulator instead of an actual MPI implementation. Your code will execute and run exactly the same, and your application won't even know the difference. You now need a parameter file with information like:

```
node {
 app1 {
  name = send_recv
  launch_cmd = aprun -n 2
  argv = 20
 }
}
```

Rather than launching your code using `mpirun` or similar, you put all your command line parameters into a `parameters.ini` file. The simulator then executes your application exactly as if you had been a real system and run:

```
shell> aprun -n 2 ./send_recv 20
```

Things get more complicated when you bring skeletonization into play. The use case above is *emulation*, exactly reproducing MPI functionality. In *skeletonization* or *simulation*, SST/macro will mimic as closely as possible the original application, but avoids as much computation and as much memory allocation as possible. This allows packing in as many simulated (virtual) MPI ranks as possible into your single `sstmac` process.

1.3 What To Expect In The User's Manual

This user's manual is mainly designed for those who wish to perform experiments with *new* applications using *existing* hardware models. This has been the dominant use case and we therefore classify those doing application experiments as "users" and those making new hardware models "developers." Getting applications to run in SST/macro should be very straightforward and requires no knowledge of simulator internal code. Making new hardware models is much more in depth and requires learning some basics of core simulator code. Those interested in making new hardware models should consult the developer's manual in the top-level source directory.

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`. You can download the git repository directly:

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

or for ssh

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

or you can download a release tarball from `https://github.com/sstsimulator/sst-macro/releases`.

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).
- 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 >=4.8.5 and clang >= 3.7 are known to work.
- (optional) OTF2: 2.0 or later for OTF2 trace replay.
- (optional) VTK 8.1 or later for creating Exodus files for traffic visualization

- (optional) Paraview 5.0 or greater for visualizing Exodus files
- (optional) Doxygen and Graphviz are needed to build the source code documentation.
- (optional) KCacheGrind or QCacheGrind to display call graphs
- (optional) Clang development libraries to enable SST source-to-source compiler

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. SST/macro can also be compiled in standalone mode that uses its own simulation core.

The workflow for installing and running on the main SST core is:

- 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 (`SST_LIB_PATH`)
- Run the `pysstmac` wrapper Python script that runs SST/macro-specific parameters OR
- Write a custom Python script

The workflow for installing and running on the standalone SST/macro core:

- Build and install SST/macro standalone to generate `sstmac` executable
- Run `sstmac` with `*.ini` parameter files

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.

Build SST/macro element library

If using the repo (not a release tarball), go to the top-level source directory and run:

```
top-level-src> ./bootstrap.sh
```

This sets up the configure script. For builds, we recommend building outside the source tree:

```
sst-macro> mkdir build
sst-macro> cd build
sst-macro/build> ../configure --prefix=$PATH_TO_INSTALL
```

If wanting to use SST core instead of the macro standalone build, run:

```
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 SST core.

SST/macro can still be built in standalone mode for features that have not been fully migrated to the SST core. This includes numerous statistics which are not yet supported by SST core. The installation and running are the same for both modes - simply remove the `--with--sst-core` parameter. A complete list of options for building can be seen by running `../configure --help`. For autoconf, options are generally divided into flags for 3rd party dependencies (`-with-X=`) and flags enabling or disabling features (`--enable-X`). Some common options are:

- –with-otf2[=location]: Enable OTF2 trace replay, requires a path to OTF2 installation.
- –with-clang[=location]: Enable Clang source-to-source tools by pointing to Clang development libraries
- –(dis|en)able-call-graph: Enables the collection of simulated call graphs, which can be viewed with KCacheGrind
- –(dis|en)able-multithread: Enables a multi-threaded backend

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 open an issue on the github page at `https://github.com/sstsimulator/sst-macro/issues` or contact SST/macro support for help (sst-macro-help@sandia.gov).

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 check
sst-macro/build> make install
sst-macro/build> make 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++` or `sstcc` compiler wrappers that are installed there for convenience to figure out where headers and libraries are. When making your skeletons and components, make sure your path is properly configured.

2.1.5 GNU pth for user-space threading: DEPRECATED

By default, Linux usually ships with `ucontext` which enables user-space threading. Mac OS X previously required an extra library be installed (GNU pth). These are no longer required and are deprecated in favor of fcontext, which is now integrated with the SST/macro distribution (see below).

For those still wishing to use pth, GNU pth is easy to download and install from source. Even easier is MacPorts.

```
shell> sudo port install pth
```

MacPorts installed all libraries to `/opt/local`. When configuring, simply add `--with-pth=$PATH_TO_PTH` as an argument. For MacPorts installation, this means configuring SST/macro with:

```
../configure --with-pth=/opt/local
```

2.1.6 fcontext

The fcontext library for user-space threading is now integrated directly with the SST/macro distribution. This provides much greater performance than GNU pth or standard Linux ucontext. Users may see as much as a 20% improvement in simulator performance. fcontext should be activated by default.

2.1.7 Known Issues

- Any build or runtime problems should be reported to sst-macro-help@sandia.gov. 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. The build system should automatically fix Ubuntu flags. If still having issues, make sure that '-Wl,–no-as-needed' is 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 sst-macro-help@sandia.gov and report the problem.

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 its own library within the SST/macro source tree by changing to `sst-macro/sst-dumpi`, where you can change its configuration options.

DUMPI can also be used as stand-alone tool (*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
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 Building Clang source-to-source support

To enable Clang source-to-source support it is not sufficient to have a Clang compiler. You have to install a special libTooling library for Clang.

2.3.1 Building Clang libTooling

The Easy Way: Mac OS X

Using MacPorts on OS X, it is trivial to obtain a Clang installation that includes libTooling:

```
port install clang-devel
```

MacPorts will place the Clang compilers in `/opt/local/bin`. Enable the devel version of Clang with:

```
port select --set clang mp-clang-devel
```

The Clang libraries will be placed into `/opt/local/libexec/llvm-devel/lib`, so the appropriate option to the sst-macro configure script is `--with-clang=/opt/local/libexec/llvm-devel`.

The Hard Way

For operating systems other than OS X, building Clang support has a few steps (and takes quite a while to build), but is straightforward. Instead of having an all-in-one tarball, you will have to download several different components. You can install more if you want build libc++, but these are not required. Obtain the following from `http://releases.llvm.org/download.html`.

- LLVM source code
- Clang source code
- libc++ source code
- libc++abi source code
- compiler-rt source code
- (optional, not recommended) OpenMP source code

Setting up the folders can be done automatically using the `setup-clang` script in `bin/tools` folder in sst-macro. Put all of downloaded tarballs in a folder, e.g. `clang-llvm`. Then run `setup-clang` in the directory. It will automatically place files where LLVM needs them. LLVM is the "driver" for the entire build. Everything else is a subcomponent. The setup script places each tarball in the following subfolders of the main LLVM folder

- tools/clang
- projects/compiler-rt
- projects/libcxx
- projects/libcxxabi
- projects/openmp

Using CMake (assuming you are in a build subdirectory of the LLVM tree), you would run the script below to configure. You no longer need to use Clang to build Clang. For the most stable results, though, you should a pre-existing Clang compiler to build the Clang development libraries.

```
cmake ../llvm \
  -DCMAKE_CXX_COMPILER=clang++ \
  -DCMAKE_C_COMPILER=clang \
  -DCMAKE_CXX_FLAGS="-O3" \
  -DCMAKE_C_FLAGS="-O3" \
  -DCMAKE_INSTALL_PREFIX=$install
```

To build a complete LLVM/Clang run:

```
cmake ../llvm \
  -DCMAKE_CXX_COMPILER=clang++ \
  -DCMAKE_C_COMPILER=clang \
  -DCMAKE_CXX_FLAGS="-O3" \
  -DCMAKE_C_FLAGS="-O3" \
  -DLLVM_ENABLE_LIBCXX=ON \
  -DLLVM_TOOL_COMPILER_RT_BUILD=ON \
  -DLLVM_TOOL_LIBCXXABI_BUILD=ON \
  -DLLVM_TOOL_LIBCXX_BUILD=ON \
  -DCMAKE_INSTALL_PREFIX=$install
```

On some systems, linking Clang might blow out your memory. If that is the case, you have to set `LD=ld.gold` for the linker. Run `make install`. The libTooling library will now be available at the `$install` location.

Any compiler used for SST (g++, icpc, clang++) can generally be mixed with most versions of the libtooling source-to-source library. NOTE: The same compiler used to build SST must have been used to build the libtooling library. However, the table below contains versions that are recommended or approved and which combinations are untested (but may work).

Compiler to build SST	Libtooling version
Clang 4,5,6	4,5,6
Clang 7,8	7,8
GCC 4.8-6	4-7
GCC 7-	?
?	8

2.3.2 Building SST/macro with Clang

Now that clang is installed, you only need to add the configure flag `--with-clang` pointing it to the install location from above. You must use the same Clang compiler to build SST that you used to build libTooling.

```
../configure CXX=clang++ CC=clang --with-clang=$install
```

Clang source-to-source support will now be built into the `sst++` compiler. If Clang development libraries are available in the default system path (as is often the case with LLVM models, e..g `module load llvm`), then you can just put `--with-clang`.

2.4 Building with OTF2

OTF2 is a general purpose trace format with specialized callbacks for the MPI API. OTF2 traces are generated by programs compiled with Score-P compiler wrappers. SST/macro 8.1 supports OTF2 trace replay and OTF2 trace generation when configured with

```
./configure --with-otf2=<OTF2-root>
```

where the OTF2 root is the installation prefix for a valid OTF2 build. OTF2 can be obtained from the Score-P project at http://www.vi-hps.org/projects/score-p. Detailed build and usage instructions can be found on the website.

2.5 Running an Application

2.5.1 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

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

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

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

since there is no Python setup involved.

2.5.2 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> sstmac -f parameters.ini --exe=./runsstmac
```

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 an importable library. The importable library contains hooks for loading a skeleton app into the simulator. NOTE: `runsstmac` appears to be an executable, but is actually built as a shared library. If using a regular compiler (e.g. gcc), the Makefile would produce an executable `runsstmac`. To ensure that building apps for SST require no modification to an existing build system, SST simple builds a shared library `runsstmac` rather than requiring renaming to the standard convention `librunsstmac.so`.

2.5.3 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.

2.5.4 Command-line arguments

There are few common command-line arguments 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.
- –otf2: If you are in a folder with all the OTF2 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 2.7
- -p [parameter]=[value]: Setting a parameter value (overrides what is in the parameter file)
- -c: If multithreaded, give a comma-separated list (no spaces) of the core affinities to use - see Section 2.6.2

2.6 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 will be discouraged as parallel simulations should use the unified SST core. However, near-term, hybrid modes and other optimizations are not fully supported in the unified SST core. The standalone core may still be required for certain cases.

2.6.1 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.

2.6.2 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.

2.6.3 Warnings for Parallel Simulation

- 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 MPI ranks.

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.

2.7 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
    router
        print all operations occurring in the router
     ....
```

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

```
debug = mpi
```

listing all flags you want separated by spaces.

Chapter 3

Basic Tutorials

3.1 SST/macro Parameter files

A minimal parameter file setting up a 2D-torus topology is shown below. An equivalent Python input file that reads an ini file is also shown. A detailed listing of parameter namespaces and keywords is given in Section 9. Both the `ini` files and Python files make careful use of namespaces.

```
amm_model = amm1
congestion_model = LogP
node {
 #run a single mpi test
 app1 {
  indexing = block
  allocation = first_available
  launch_cmd = aprun -n8 -N1
  name = sstmac_mpi_testall
  argv =
  sendrecvMessage_size = 128
 }
 ncores = 1
 memory {
  model = simple
  bandwidth = 1GB/s
  latency = 10ns
 }
 proc {
  frequency = 1GHz
 }
 nic {
  injection {
   bandwidth = 1GB/s
   latency = 1us
  }
  model = simple
 }
}
switch {
 link {
  bandwidth = 1.0GB/s
   latency = 100ns
 }
 logp {
   bandwidth = 1GB/s
```

```
    out_in_latency = 1us
  }
}

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

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`. If inside a given namespace, SST/macro looks only within that namespace.

The preferred syntax more closely resembles C++ namespace declarations. Namespaces are 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. A detailed listing of parameter namespaces and keywords is given in Section 9.

3.1.2 Initial Example

Continuing with the example above, we see the input file is broken into namespace sections. First, application launch parameters for each node 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 `node.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 for the node, network interface controller (NIC), and memory.

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
node {
 app1 {
  indexing = block
  allocation = first_available
  launch_cmd = aprun -n2 -N1
  name = user_mpiapp_cxx
  argv =
  # Application parameters
  sendrecvMessage_size = 128
 }
}
```

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

3.2 SST Python Files

For SST core, SST/macro provides a Python translator for the ini files to a Python input deck. For those wishing to directly use Python inputs (or understand better SST core files), an example Python file is included here to illustrate using Macro. Additionally, you can look at `sstmac/sst_core/sstmacro.py` to see more details of building and linking SST components.

SST/macro provides a particular idiom for setting up systems. Rather than directly instantiate components, `sstmacro.py` provides an interconnect object. Parameter namespaces are created through nested dictionaries. One all parameters are created in the dictionaries, all the components are build and run through the `ic.build()` command.

```
import sst
import sst.macro
from sst.macro import Interconnect

link_bw="12.5GB/s"
link_lat="100ns"
logpParams = {
  "bandwidth" : "12.5GB/s",
  "hop_latency" : "100ns",
  "out_in_latency" : "1.2us",
}

swParams = {
  "name" : "pisces",
  "mtu" : "4KB",
  "arbitrator" : "cut_through",
  "router" : {
    "name"     :     "torus_minimal",
  },
  "link" : {
    "bandwidth" : link_bw,
    "latency" : link_lat,
    "credits" : "128KB",
  },
  "xbar" : {
    "bandwidth" : "1000GB/s",
    "latency" : "10ns",
  },
  "logp" : logpParams,
}

mpiParams = {
```

```
 "max_vshort_msg_size" : 4096,
 "max_eager_msg_size" : 64000,
 "post_header_delay" : "0.35906660us",
 "post_rdma_delay" : "0.88178612us",
 "rdma_pin_latency" : "5.42639881us",
 "rdma_page_delay" : "50.50000000ns",
}

appParams = {
  "allocation" : "hostname",
  "indexing" : "hostname",
  "exe" : "halo3d-26",
  "argv" : "-pex 4 -pey 4 -pez 4 -nx 128 -ny 128 -nz 128 -sleep 0 -iterations 10",
  "launch_cmd" : "aprun -n 64 -N 1",
  "allocation" : "first_available",
  "indexing" : "block",
  "mpi" : mpiParams,
}

memParams = {
 "name" : "pisces",
 "latency" : "10ns",
 "total_bandwidth" : "100GB/s",
 "max_single_bandwidth" : "11.20GB/s",
}

nicParams = {
  "name" : "pisces",
  "injection" : {
    "mtu" : "4KB",
    "redundant" : 8,
    "bandwidth" : "13.04GB/s",
    "arbitrator" : "cut_through",
    "latency" : "0.6us",
    "credits" : "128KB",
  },
  "ejection" : {
    "latency" : link_lat,
  },
}

nodeParams = {
  "memory" : memParams,
  "nic" : nicParams,
  "app1" : appParams,
  "name" : "simple",
  "proc" : {
    "frequency" : "2GHz",
    "ncores" : "4",
  }
}

topoParams = {
 "name" : "torus,
 "geometry" : "[4,4,4]",
}

params = {
  "node" : nodeParams,
  "switch" : swParams,
  "topology" : topoParams,
}

ic = Interconnect(params)
ic.build()
```

Again, to see all the details of creating and linking components, refer to the `sstmac/sst_core/sstmacro.py` file.

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
```

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

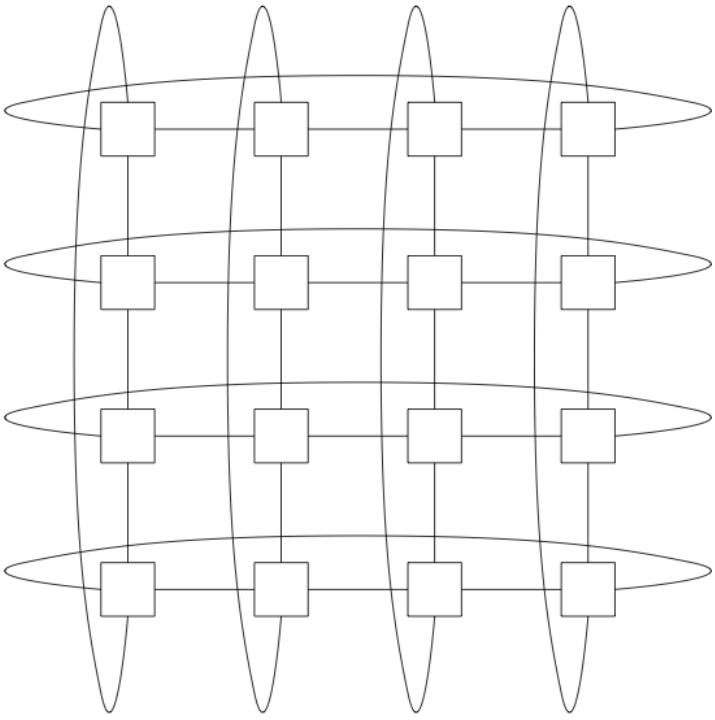


Figure 3.1: 4 x 4 2D Torus

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

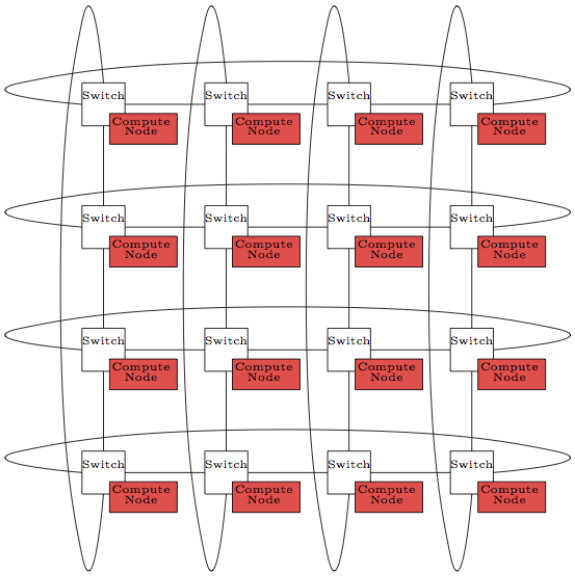


Figure 3.2: 4 x 4 2D Torus of Network Switches with Compute Nodes

"directly" connected to it. A more accurate picture of the network is given in Figure 3.2. While in many previous architectures there was generally a one-to-one correspondence between compute nodes and switches, more recent architectures have multiple compute nodes per switch (e.g. Cray Gemini with two nodes, Cray Aries with four nodes). Multiple nodes per switch can be specified via a concentration parameter:

```
topology {
 name = torus
 geometry = 4 4
 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
 geometry = 4 4
 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
 geometry = 4 4
 redundant = 1 1
}
switch.link.bandwidth = 2GB/s
```

```
topology {
 name = torus
 geometry = 4 4
 redundant = 2 2
}
switch.link.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 3.3). In going

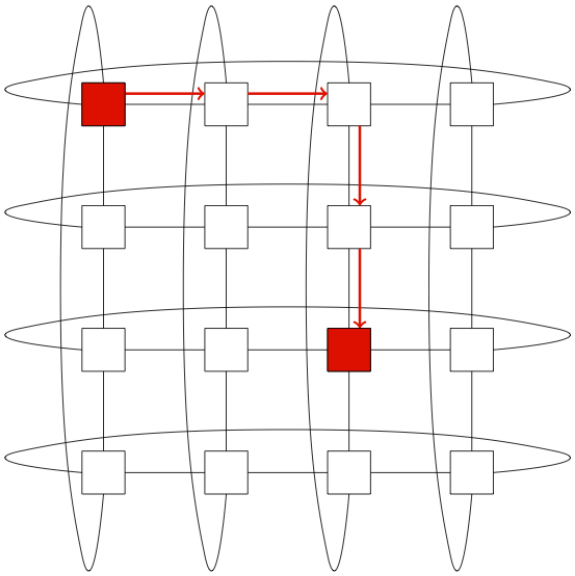


Figure 3.3: Dimension-Order Minimal Routing on a 2D Torus

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:

```
switch {
 router {
  name = min_ad
 }
}
```

which specifies minimal adaptive routing. There are now multiple valid paths between network endpoints, one of which is illustrated in Figure 3.4. At each network hop, the router chooses the *productive* path with least congestion. In some cases,

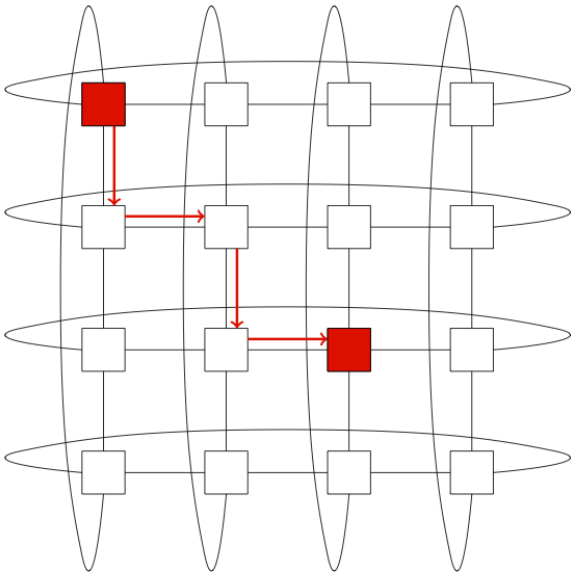


Figure 3.4: Adaptive Minimal Routing on a 2D Torus

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:

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

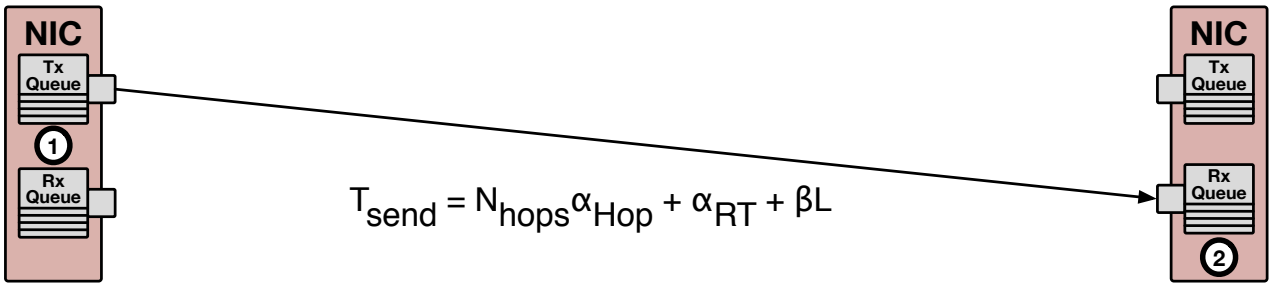


Figure 3.5: MACRELS (Messages with AnalytiC REally Lightweight Simulation) skips congestion modeling and approximates send delays using a simple latency/bandwidth estimate, similar to the LogGOP model. Modeling occurs on entire flows, rather than individual packets. For details on numbered steps, see text.

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
```

Details are shown for traffic moving from source to destination in Figure 3.5. Modeling occurs on entire flows, rather than individual packets.

1. Flows queue waiting for NIC injection link to become available. Flow is forwarded to destination NIC based after computed delay.
2. Flows queue waiting for NIC ejection link to become available. Flow finishes after ejection link becomes available.

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 (Packet-flow Interconnect Simulation for Congestion at Extreme-Scale) models individual packets moving through the network. Flits (flow-control units) are approximately modeled using flow-like approximations. Packets can have partial occupancies in several different buffers, approximating wormhole routing. However, arbitration is modeled on whole packets, not individual flits (see Figure 3.6)

1. A message (flow) is broken up into packets. Depending on available space in the Tx buffer, a limited number of packets may be able to queue up in the buffer. If credits are available in the Rx buffer for the link and the link is idle, the packet moves into the next Rx buffer after a computed delay.

2. The router selects a path for the packet and the packet requests to the crossbar to transmit to the corresponding output port. If credits are available for the Rx buffer, the crossbar may select the packet in arbitration and move it to the output buffer. After moving, the Rx buffer returns credits to the previous Tx buffer for that packet.
3. Step 1 is repeated for the next Rx buffer, waiting for credits and link availability.
4. Repeat Step 2
5. Repeat Step 3
6. Packet arrives in NIC Rx queue and queues waiting to inject into local memory. After injection, the Rx buffer returns credits to the corresponding Tx buffer.

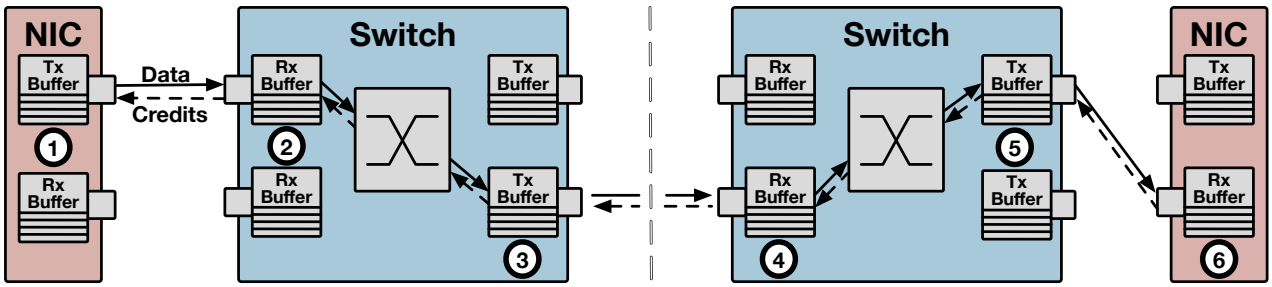


Figure 3.6: PISCES (Packet-flow Interconnect Simulation for Congestion at Extreme-Scale) models individual packets moving through the network. Flits (flow-control units) are approximately modeled using flow-like approximations. For details on numbered steps, see text.

PISCES provides two mechanisms for treating flit-level flow control discussed next.

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:

```
switch.mtu = 1024B
node.nic.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. More details are given in Figure 3.6.

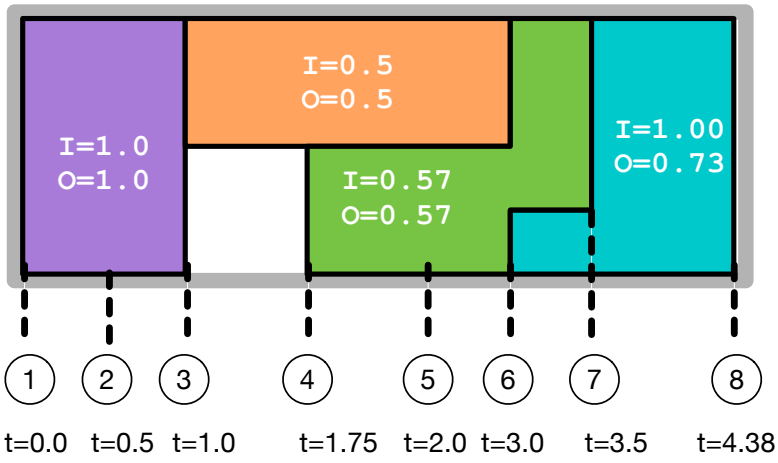


Figure 3.7: Timeline of four different packets passing through a PISCES cut-through bandwidth arbitrator. The incoming bandwidth (I) and outgoing bandwidth (O) are shown for each packet. Time is the horizontal axis. Bandwidth consumed by a packet is shown by the vertical extent of each packet. The individual 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.

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 3.7 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.

- 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 3.7) 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 SCULPIN

Under current architectural trends, switches have ample buffer space and crossbar bandwidth, making the mostly likely bottleneck edge bandwidth through the output ports. SCULPIN (Simple Congestion Unbuffered Latency Packet Interconnection Network) models the main source of contention in today's networks occurring on the output port ser/des. Unlike PISCES, individual flits are not able to wormhole route across links interspersed with flits from other packets.

1. A message (flow) is broken up into packets. Each packet waits in the queue to send based on link availability and QoS.
2. After being selected, the packets are forwarded to the switch. Packets are immediately routed to the correct output port, skipping crossbar arbitration. Packets wait in unbounded queues, thereby assuming sufficient buffer space is always available.
3. Repeat Step 1. Packet waits in queue until link becomes available based on QoS. Packet is immediately forwarded to next output port, skipping arbitration
4. Repeat Step 1.
5. Packet arrives in NIC Rx queue (no credits, buffer assumed to always have space). Packets queue waiting to inject into local memory.

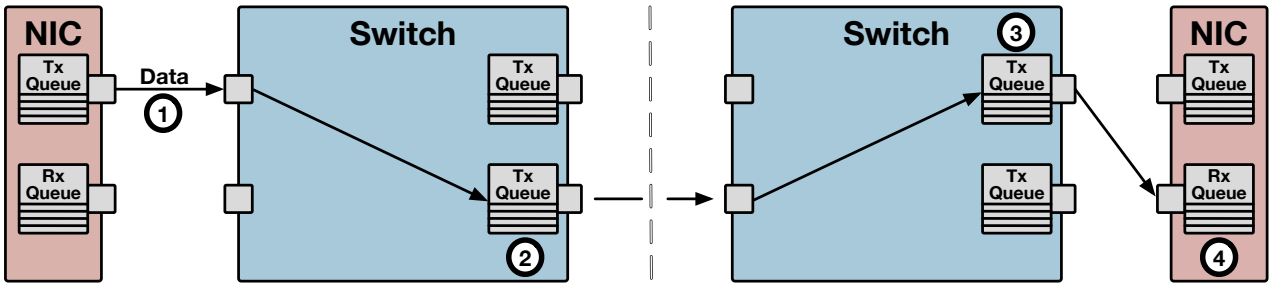


Figure 3.8: SCULPIN (Simple Congestion Unbuffered Latency Packet Interconnection Network) models the main source of contention in today's networks occurring on the output port ser/des. For details on numbered steps, see text.

3.4.4 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  int main(int argc, char **argv)
6  {
7    int message_size = 128;
8    int me, nproc;
9    int tag = 0;
10   int dst = 1;
11   int src = 0;
12   MPI_Status stat;
```

```
13
14    MPI_Init(&argc,&argv);
15    MPI_Comm world = MPI_COMM_WORLD;
16    MPI_Comm_rank(world,&me);
17    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 `userSkeletonMain`, 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

```
node {
 app1 {
  exe = <PATH_TO_EXE>
```

While MPI would have produced an executable, SST works by loading shared object files using `dlopen`. To get SST to load the skeleton, you must specify the path of the "executable" in the input file. Using `dlopen` tricks, SST finds the main function in the .so file and calls it to spawn the skeleton app. Just as an executable can only have one main, SST shared object files can only have a single executable in them at a time.

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.

For more details on what exactly the SST compiler wrapper is doing, you can specify `SSTMAC_VERBOSE=1` as an environment variable to have SST print out detailed commands. Additionally, you can specify `SSTMAC_DELETE_TEMPS=0` to examine any temporary source-to-source files.

3.5.1 DEPRECATED: App name macro

Previously, applications had to be "named" by using the `sstmac_app_name` macro. This macro can still be define just above main to give a descriptive name to the skeleton:

```
1   #include <mpi.h>
2
3   #define sstmac_app_name simple_test
4
5   int main(int argc, char **argv)
6   {
```

This was previously required for starting applications, but now is only used for providing descriptive labels for certain applications. It is not recommended for use anymore and is ignored. An equivalent naming can just be provided by:

```
node {
 app1 {
   exe = <PATH_TO_EXE>
   name = simple_test
 }
}
```

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.

```
node {
 app1 {
  name = user_mpiapp_cxx
  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)

```
node.app1.allocation = first_available
```

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).

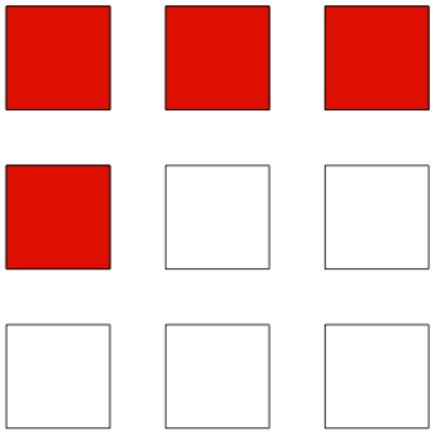


Figure 3.9: First available Allocation of 4 Compute Codes on a 3x3 2D Torus

```
app1 {
 allocation = cartesian
 cart_sizes = [2,2]
 cart_offsets = [0,0]
}
```

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.

```
node.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).

```
node.app1.indexing = block
```

In contrast, round-robin spreads out MPI ranks by assigning consecutive MPI ranks on different nodes (Figure 3.13).

```
node.app1.indexing = round_robin
```

Finally, one may also choose

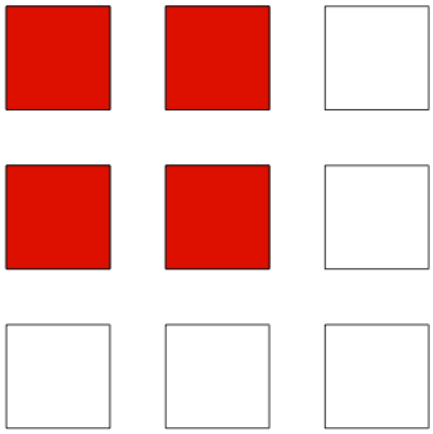


Figure 3.10: Cartesian Allocation of 4 Compute Codes on a 3x3 2D Torus

```
node.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.

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

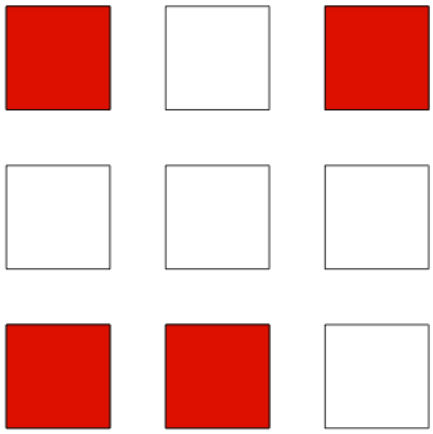


Figure 3.11: Random Allocation of 4 Compute Codes on a 3x3 2D Torus

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

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

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

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

3.8 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,

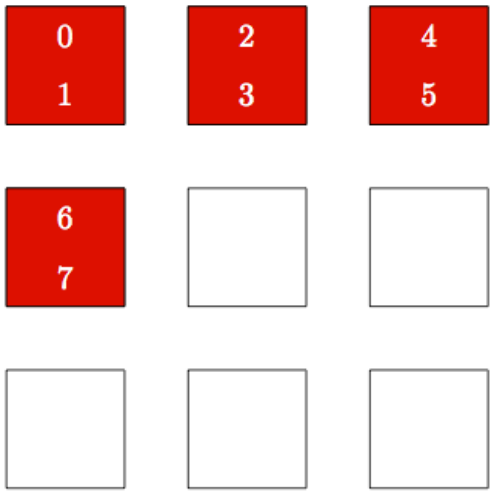


Figure 3.12: Block Indexing of 8 MPI Ranks on 4 Compute Nodes

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> ./bootstrap.sh
```

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` 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"
```

3.8.2 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

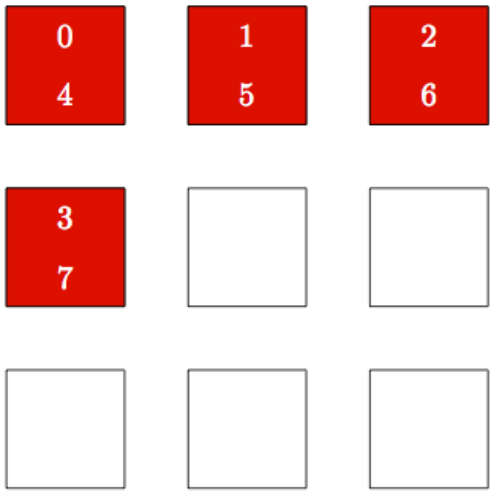


Figure 3.13: Round-Robin Indexing of 8 MPI Ranks on 4 Compute Nodes

	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 ($1\mu s$)	5)Wait for ack; block	
$t = 2\mu s$	7)Recv at NIC 1; unblock proc 1		8)Send ack for recv ($1\mu s$); block
$t = 3\mu s$	9)Deliver ack to NIC 0 ($1\mu s$) 10)Send done; unblock proc 1		11)Continue execution...
$t = 4\mu s$	12)Recv at NIC 0; unblock proc 0	13)Continue execution...	

Figure 3.14: Progression of Discrete Event Simulation for Simple Send/Recv Example

```
1 int MPI_Send(...)
2 {
3   /** Start profiling work */
4   ...
```

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

collecting profile information and then directly calling the PMPI functions.

We examine DUMPI using a very basic example program.

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

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

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

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

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

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

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

This produces the output

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

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

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

3.8.3 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.

```
node {
 app1 {
  indexing = dumpi
  allocation = dumpi
  name = parsedumpi
  dumpi_metaname = testbgp.meta
 }
}
```

We 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
 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
node {
 app1 {
```

```
  indexing = dumpi
  allocation = hostname
  name = parsedumpi
  dumpi_metaname = dumpi-2013.09.26.10.55.53.meta
  dumpi_mapname = deepthought.map
 }
}
# Machine parameters
topology {
 name = torus
 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
node {
 app1 {
  indexing = block
  allocation = first_available
  dumpi_metaname = dumpi-2013.09.26.10.55.53.meta
  name = parsedumpi
  size = 2
 }
}
# Machine parameters
topology {
 name = crossbar
 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 Using Score-P and OTF2

OTF2 is part of Score-P. Sources for both can be found here

```
http://www.vi-hps.org/projects/score-p
```

Trace collection requires both Score-P and OTF2 installations. Trace replay with SST/macro requires only OTF2.

3.9.1 Trace Collection

Score-P's default collection strategy will include every function call in the trace, making even small programs produce untenably large traces. Score-P supports collection filters, which can restrict collection at a minimum to MPI and OMP function calls. At the end of the program's runtime, traces from each rank are put in a common directory. An MPI program must be compiled with Score-P to produce traces:

```
scorep-mpicxx -o test.exe test.cc
```

To limit the size of the traces, run the program with:

```
# these environment variables are picked up by Score-P at runtime
export SCOREP_ENABLE_TRACING=true
export SCOREP_TOTAL_MEMORY=1G
export SCOREP_FILTERING_FILE='scorep.filter'

mpirun -n 2 test.exe
```

The file `scorep.filter` should contain:

```
SCOREP_REGION_NAMES_BEGIN EXCLUDE *
```

To view a plain-text representation of the trace after running, use the otf2-print tool.

```
otf2-print scorep-*/traces.otf2
```

3.9.2 Trace Replay

SST/macro will use a trace replay skeleton for OTF2 in much the same way as it does for dumpi. SST/macro trace replays configured using *.ini files.

```
...

node {
 app1 {
  otf2_timescale = 1.0
  name = otf2_trace_replay_app
  size = N
  otf2_metafile = <trace-root>/scorep-20170309_1421_27095992608015568/traces.otf2
 # debugging output
  otf2_print_mpi_calls=false
  otf2_print_trace_events=false
  otf2_print_time_deltas=false
  otf2_print_unknown_callback=false
 }
}
```

3.10 Statistics

SST/macro statistics current only work in standalone mode due to the rigid structure of statistics in SST core. Fixes are planned for SST core for the 10.0 release in 2020. Until then, most of the statistics will require the standalone core. Both standalone and SST core do follow the same basic structure, however, and the tutorial below covers concepts important to both. The three abstractions for understanding statistics are collectors, groups, and outputs. We will use the running example of a bytes sent statistic.

3.10.1 Collectors

Statistic collectors inherit from `Statistic<T>` and are used by components to collect individual statistics. Statistics are collected through the `addData` function:

```
1 xmit_bytes_->addData(pkt->byteLength());
```

The statistic only specifies the *type* of data collected, not the manner of collection (histogram, accumulate). In this case, e.g., we would have:

```
1 Statistic<uint64_t>* xmit_bytes_;
```

Statistic objects generally do three things, with the exception of special custom statistics.

- Collect data through an *addData* function.
- Register output fields before collecting data through the `registerField` function.
- Pass along fields to output when done collecting data through the `outputField` function.

The data *collected* and the data *output* are not the same - and may not even be of the same type. For an accumulator, data collected and data output are the same (sum of collected values). For a histogram, data collected is, e.g., a single integer, while the data output is the counts in a series of bins. If there are ten bins, the histogram will register ten output fields at the beginning and output ten fields at the end.

3.10.2 Outputs

Statistic outputs provide an abstract interface for outputting fields. In SQL or Pandas terminology, each statistic defines a row in a table. Each field will be a column. Outputs therefore map naturally to a CSV file. SST core also provides an HDF5 output, for those statistics that work with SST core.

3.10.3 Groups

Statistics can be grouped together when appropriate, for example bytes sent statistics for switches may want to be grouped together. Each group is associated with a given output. Most commonly, a group is only used to link statistics to the same output. However, aggregation of statistics can potentially be performed as well for certain cases.

3.10.4 SST/macro Standalone Input

Each statistic has a name, which specifies a parameter namespace in the parameter file. In the case above, we activate an “xmit_bytes" statistic.

```
nic {
  injection {
   xmit_bytes {
     type = accumulator
     output = csv
     group = test
   }
  }
}
```

As discussed above, we must specify the type of collection, the type of output, and the group name. The output file name (test.csv) is generated from the group name. The CSV output looks like:

```
name,component,total
xmit_bytes,nid0,16952064
xmit_bytes,nid1,6635264
xmit_bytes,nid2,7542016
```

For a histogram with 5 bins, we could specify an input:

```
 xmit_bytes {
  type = histogram
  output = csv
  group = all
  num_bins = 5
  min_value = 0
  max_value = 5KB
}
```

which then generates columns for the bins, e.g.

```
name,component,numBins,binSize,bin0,bin1,bin2,bin3,bin4
xmit_bytes,nid0,5,1000,1235,559,396,31,3746
xmit_bytes,nid1,5,1000,870,439,322,11,1292
xmit_bytes,nid2,5,1000,838,396,279,11,1551
```

3.10.5 Custom Statistics

Certain statistics (examples below) do not fit into the model of row/column tables and require special `addData` functions. Rather than declare themselves as `Statistic<T>` for some numeric type T, they declare themselves as `Statistic<void>` and have a completely custom collection and output mechanism. They also generally do not rely an abstract interfaces. For example, a bytes sent statistic output can choose a CSV histogram or an HDF5 accumulator or a text list. Here, the statistic can only have a single type and works with a specific output.

3.11 OTF2 Trace Creation

SST/macro can emit OTF2 traces from MPI simulations, if compiled with:

```
build> ../configure --enable-otf2=$PATH_TO_OTF2
```

This is an example of a custom statistic, discussed in 3.10.5. This gets activated by:

```
...

node {
 app1 {
  mpi {
    otf2 {
      type = otf2writer
      group = app1
      output = otf2
    }
  }
 }
}
```

The statistic is given to the MPI namespace. Good practice is generally to name the group after the app.

3.12 Call Graph Visualization

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

```
build> ../configure --prefix=$INSTALL_PATH --enable-call-graph
```

The extra flag enables special instrumentation. In the default build, the instrumentation is not added to avoid overheads. 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 must be activated for a given application.

```
node {
  app1 {
    call_graph {
      type = call_graph
      output = cachegrind
      group = test
    }
  }
}
```

After running the above, a `test.callgrind.out` file should appear in the folder based on the group name. Additionally, a summary CSV file - `test.csv` or other group name - is generated with various aggregate statistics.

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).

In the summary file, each function is broken down into "self" time (time spent directly in the function itself) and time spent in various subroutines.

```
app1.rank0.thread0,main,self,40000000000000
app1.rank0.thread0,main,sleep,20000000000000000
app1.rank0.thread0,main,MPI_Init,101362856608
app1.rank0.thread0,main,MPI_Finalize,40013333328
app1.rank0.thread0,main,MPI_Allgather,344587523048
app1.rank0.thread0,main,MPI_Alltoall,80969809408
app1.rank0.thread0,main,MPI_Allreduce,180006666664
app1.rank0.thread0,main,MPI_Barrier,320740095096
app1.rank0.thread0,main,MPI_Gather,160234285648
app1.rank0.thread0,main,MPI_Reduce,161223333064
....
```

Here is a breakdown of a benchmark for the `main` routine. A small amount of self-time is shown, along with the time spent in other routines. Each of these subroutines themselves can also have subroutines:

```
app1.rank0.thread0,MPI_Allgather,self,121499999400
app1.rank0.thread0,MPI_Allgather,memcopy,3034190336
```

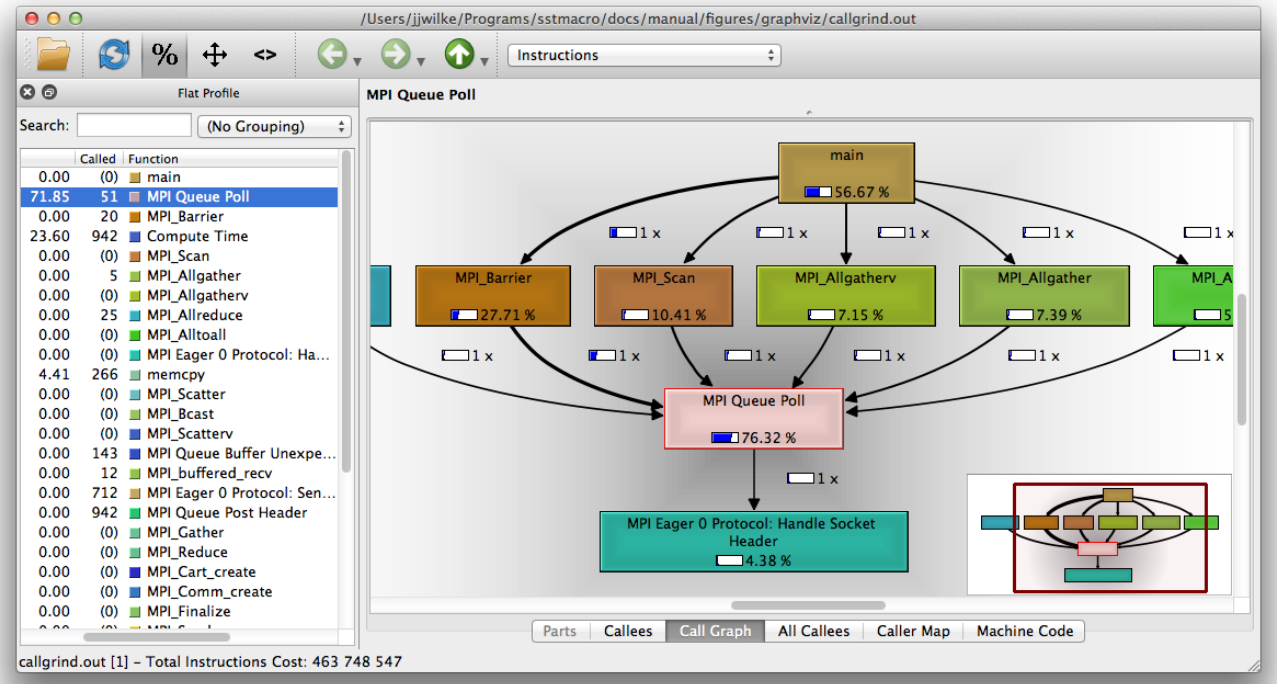


Figure 3.15: QCachegrind GUI

3.13 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 3.18). 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. The most commonly used are the MPI spyplots, for which you activate the spyplot as part of the MPI subcomponent.

```
node {
  app1 {
    mpi {
      spy_bytes {
        type = spyplot
        output = csv
        group = app1
      }
    }
  }
}
```

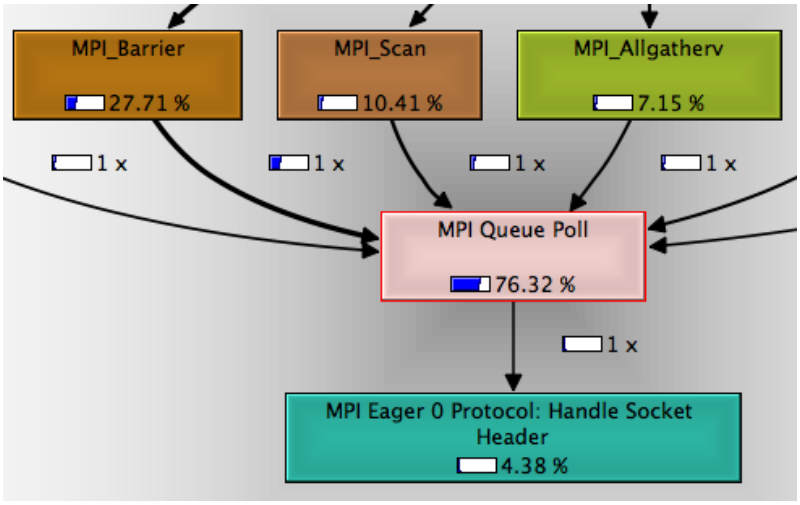


Figure 3.16: QCachegrind Call Graph of MPI Functions

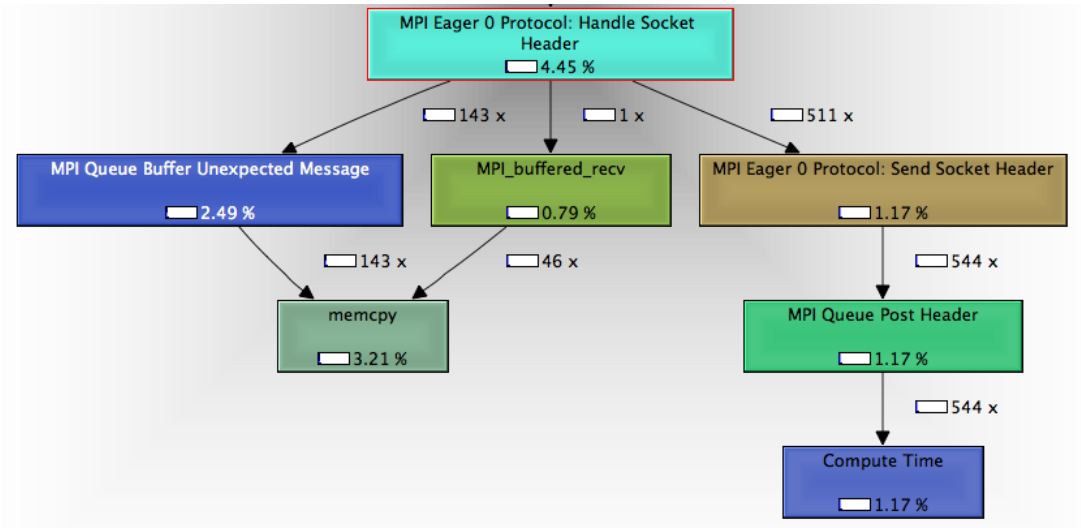


Figure 3.17: QCachegrind Expanded Call Graph of Eager 0 Function

After running, there will be a CSV containing the data sent by each component to another component. The same statistic can be activated in both the `node.app1.mpi` namespaces and the `node.nic` namespaces. The type of the statistic must be spyplot, but the output can be other formats (but just use csv).

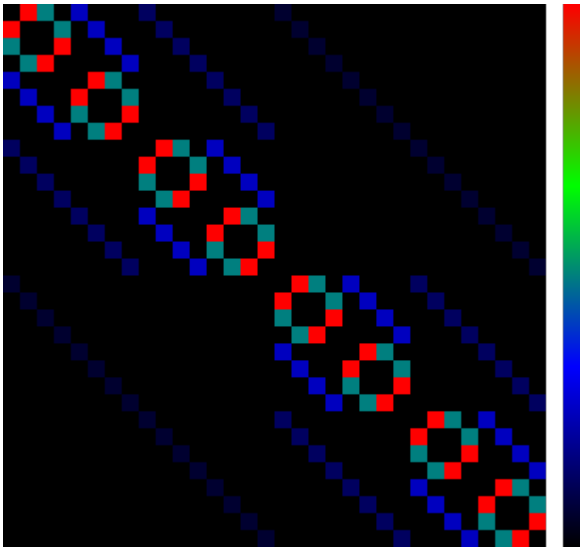


Figure 3.18: Spyplot of Bytes Transferred Between MPI Ranks for MPI_Allreduce

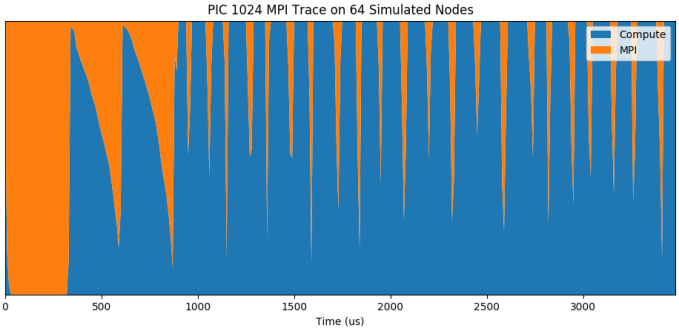


Figure 3.19: Application Activity (Fixed-Time Quanta; FTQ) histogram

3.14 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 critical code regions, they lack temporal information. Figure 3.19 displays the proportion of time spent by ranks in MPI communication and computation in a PIC trace replay with respect to simulation time. After running, two new files appear in the folder: `<fileroot>_app1.py` and `<fileroot>_app1.dat` that can use Python's matplotlib to generate plots. Previously, plots were generated using Gnuplot, but this has been deprecated in favor of much more aesthetically pleasing maplotlib output.

```
your_project # python output_app1.py --help
usage: output_app1.py [-h] [--show] [--title TITLE] [--eps] [--pdf] [--png]
                      [--svg]
```

```
optional arguments:
  -h, --help      show this help message and exit
  --show          display the plot on screen
  --title TITLE   set the title
  --eps           output .eps file
  --pdf           output .pdf file
  --png           output .png file
  --svg           output .svg file
```

Generating the histogram requires matplotlib, and visualizing the histogram interactively with `--show` requires a screen or X11 forwarding. FTQ aggregates tags into tunable time "epochs". An epoch states the ratio of each tag represented at a point in time. Larger epochs will smooth the graph and decrease the quantity of data required to render a plot; while a smaller epoc will add more detail, at the risk of making the plot excessively volatile.

SST/macro activates FTQ for a given application as:

```
node {
 app1 {
  name = sstmac_mpi_testall
  launch_cmd = aprun -n 8 -N 2
  ftq {
   type = ftq_calendar
   epoch_length = 1ms
   output = ftq
   group = app1
  }
  print_times = false
  message_size = 400B
 }
}
```

where the `fileroot` a path and a file name prefix.

3.15 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. These are based on the XmitWait and XmitBytes performance counters from OmniPath, and aim to provide similar statistics as those from production systems.

3.15.1 XmitBytes

To active a message size histogram on the NIC or the switches to determine the data sent by individual packets, the parameter file should include, for example:

```
node {
 nic {
  injection {
   xmit_bytes {
     output = csv
     type = accumulator
     group = test
   }
  }
 }
}
```

In contrast the custom statistics above, this is a row-table statistic that can have different types and outputs. The same stat can be activated in both the `node.nic.injection` and `switch` namespaces.

3.15.2 XmitWait

To estimate congestion, SST/macro provides an `xmit_wait` statistic. This counts the total amount of time spent in stalls due to lack of credits. The time is accumulated and reported in seconds.

```
node {
 nic {
  injection {
   xmit_wait {
     output = csv
     type = accumulator
     group = test
   }
  }
 }
}
```

The same stat can be activated in both the `node.nic.injection` and `switch` namespaces.

3.15.3 XmitFlows

The previous statistics track the bytes sent by packets and as such are agnostic to the messages (or flows) sending them. If interested in the flow-level statistics, it can be activated as:

```
node {
 app1 {
  mpi {
    xmit_flows {

    }
  }
 }
}
```

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 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 Cascade, 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
 geometry = 4 4 4
}
node {
 app1 {
  launch_cmd = aprun -n 8
  indexing = block
  allocation = cartesian
  cart_sizes = [2,2,2]
  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
 geometry = 4 4 4
 concentration = 2
}
app1 {
 launch_cmd = aprun -n 8
 indexing = block
 allocation = cartesian
 cart_sizes = [2,2,1,2]
 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.

4.2 Hypercube

Although never used at scale in a production system, the generalized hypercube is an important topology to understand, particularly for flattened butterfly and Cascade. 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.

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).

4.2.1 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:

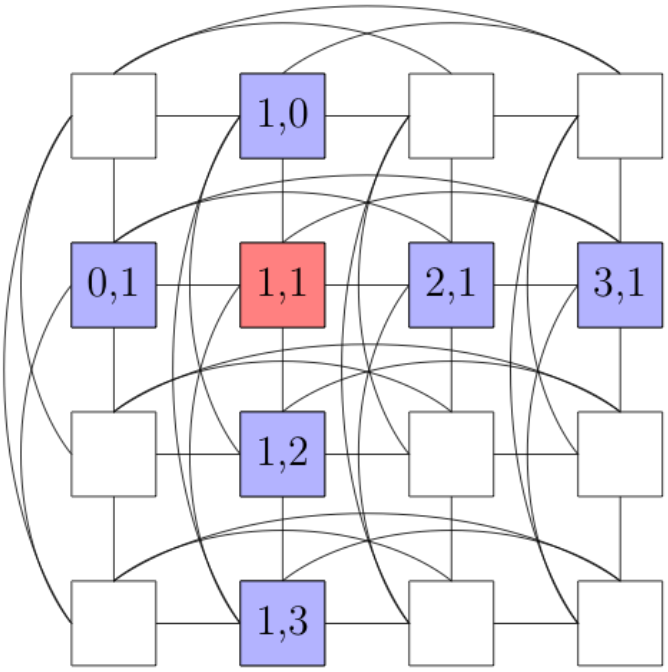


Figure 4.1: Hypercube with links and connections within a row/column

```
node {
 app1 {
  launch_cmd = aprun -n 5
  indexing = coordinate
  allocation = coordinate
  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

4.2.2 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.

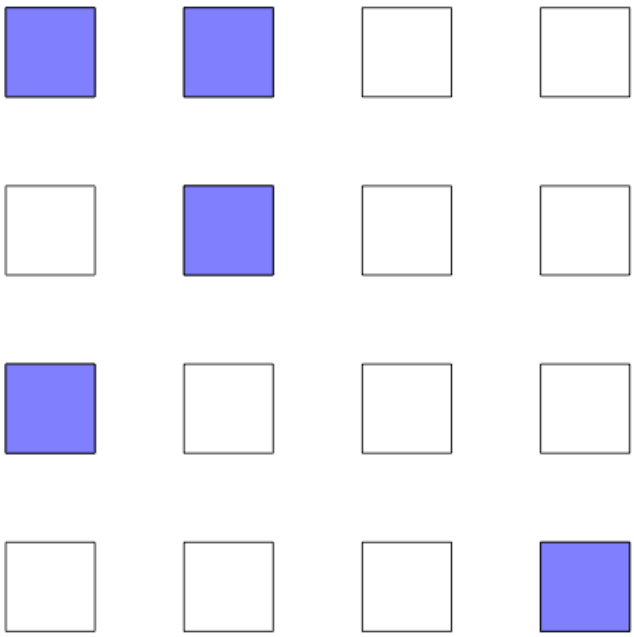


Figure 4.2: Hypercube allocation for given set of coordinates

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 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.

4.3 Fat Tree

SST provides a very flexible fat-tree topology which allows both full bandwidth and tapered bandwidth configurations using either uniform or non-uniform switches. This flexibility requires a farily complicated set of input parameters which are best introduced by examining a couple of example configurations. Consider the full-bandwidth topology in Figure 4.5 which uses uniform 8-port switches throughout.

The SST fat-tree is strictly a 3-level topology, with the switch levels refered to as leaf (bottom), aggregation (middle), and core (top). Interconnected leaf and aggregation switches form an aggregation subtree, which forms the basic unit of a fat-tree topology. The structure of the aggregation subtree is, itself, flexible and places few constraints on the number of subtrees or the way they are connected to the core level. In Figure 4.5, there are 4 leaf switches and 4 aggregation switches per subtree, and each leaf switch has a concentration of four nodes per switch. Balancing bandwidth, there are 4 ports going up from each leaf switch and 4 ports going down from each aggregation switch. This subtree can be specified as follows:

```
topology.leaf_switches_per_subtree = 4
```

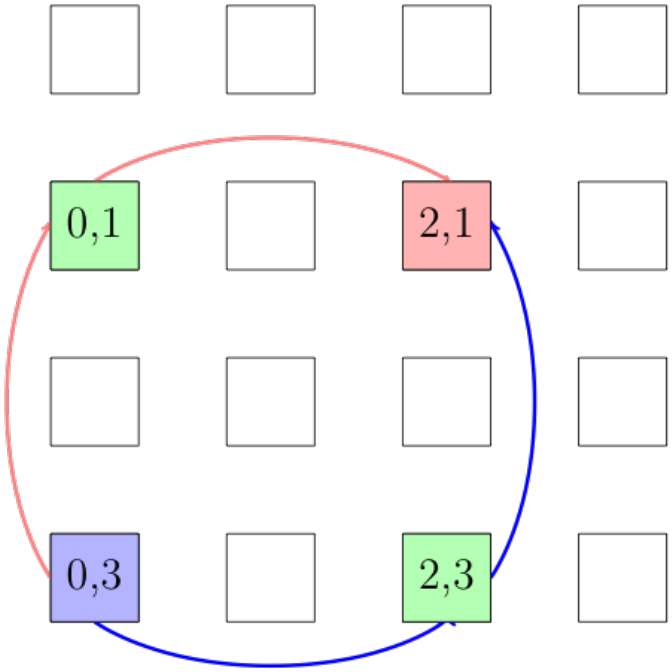


Figure 4.3: Minimal routing within a hypercube showing path diversity. Packet travels from blue to red, passing through green intermediate switches.

```
topology.agg_switches_per_subtree = 4
topology.concentration = 4
topology.up_ports_per_leaf_switch = 4
topology.down_ports_per_agg_switch = 4
```

In this example we have 2 aggregation subtrees. There are four ports going up from each aggregation switch. All of the ports on the core switches go down, so the number of core switches required (4) is only half the number of total aggregation switches (8). This core configuration can be specified as follows:

```
topologies.num_agg_subtrees = 2
topologies.num_core_switches = 4
topologies.up_ports_per_agg_switch = 4
topologies.down_ports_per_core_switch = 8
```

Putting it all together with the topology name results in:

```
topology.name = fat_tree
logy.leaf_switches_per_subtree = 4
topology.agg_switches_per_subtree = 4
topology.concentration = 4
topology.up_ports_per_leaf_switch = 4
topology.down_ports_per_agg_switch = 4
topologies.num_agg_subtrees = 2
topologies.num_core_switches = 4
topologies.up_ports_per_agg_switch = 4
topologies.down_ports_per_core_switch = 8
```

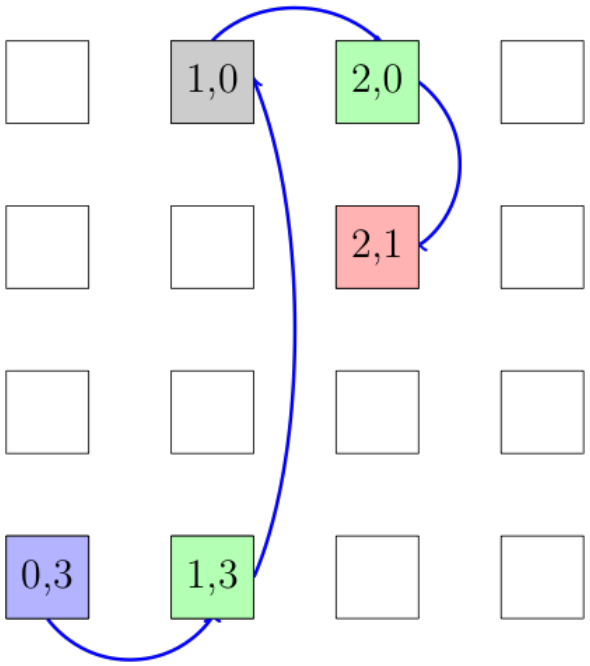


Figure 4.4: Valiant routing within a hypercube. Packet travels from blue to red via a random intermediate destination shown in gray. Additional intermediate switches are shown in green.

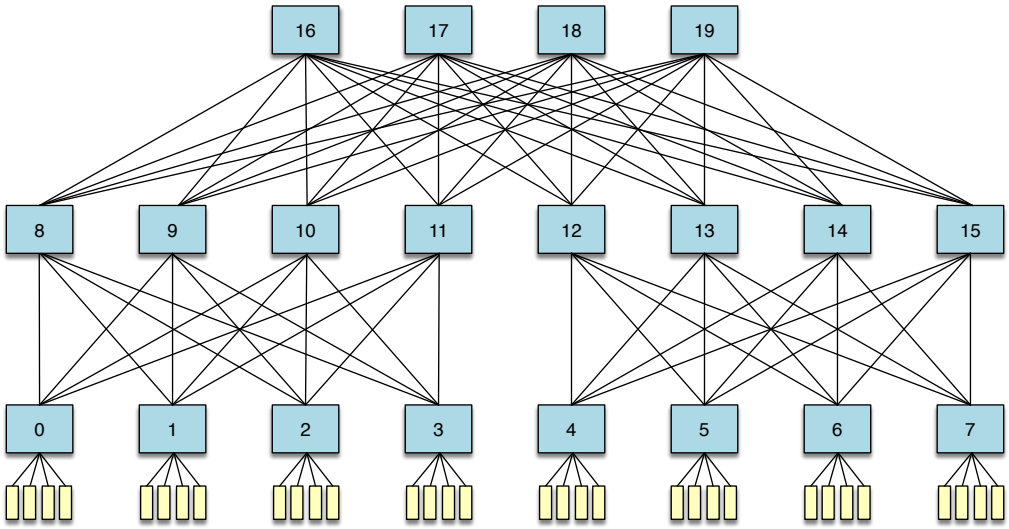


Figure 4.5: Full-bandwidth fat-tree topology using uniform 8-port switches.

The next example, though somewhat contrived, better demonstrates the fat-tree input flexibility. Suppose that one wanted to use the same 8-port switches to construct a 3-level fat-tree that was both cheaper and had more endpoints (nodes), at the cost of interswitch bandwidth. One possible configuration is shown in Figure 4.6.

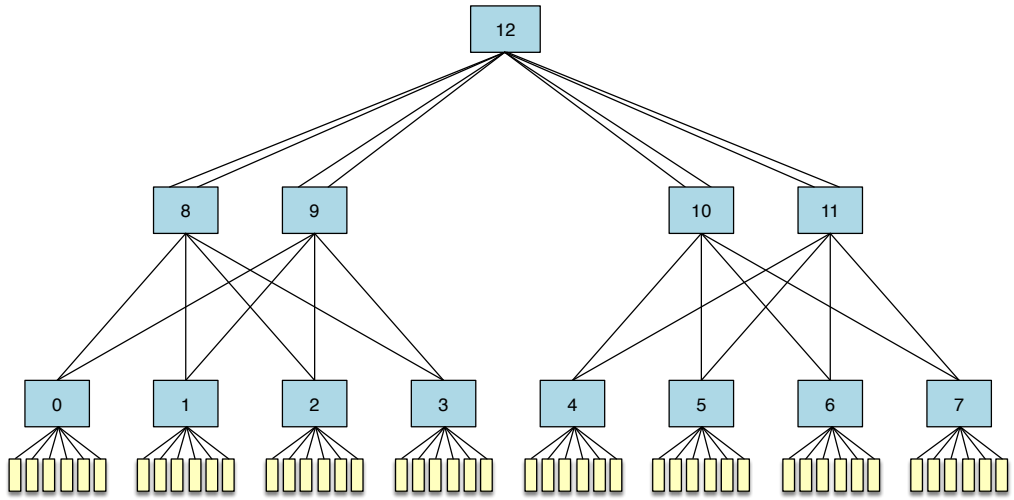


Figure 4.6: A tapered fat-tree topology using uniform 8-port switches.

Here the concentration has been increased to 6 nodes per leaf switch, leaving only two up ports per leaf switch. Thus an aggregation subtree has a total of only 8 leaf up ports, which requires at least two aggregation switches (in order to have any ports left to connect into the core). Each aggregation switch is then required to have 4 ports heading down. The subtree can be configured as follows:

```
topology.leaf_switches_per_subtree = 4
topology.agg_switches_per_subtree = 2
topology.concentration = 6
topology.up_ports_per_leaf_switch = 2
topology.down_ports_per_agg_switch = 4
```

There are a total of four aggregation switches. If the bandwidth is allowed to taper again, a single 8-port core switch can accomodate 2 ports coming up from each aggregation switch. This core configuration can be specified as follows:

```
topologies.num_agg_subtrees = 2
topologies.num_core_switches = 1
topologies.up_ports_per_agg_switch = 2
topologies.down_ports_per_core_switch = 8
```

This is a heavily tapered tree and also has the downside of using only 6 ports per switch in the aggregation level. This example was chosen more for its illustrative rather than practical value, though there are certainly applications where it would be perfectly adequate. More practical tapering becomes an option when you increase the number of ports per switch, but visualizations become more difficult to grasp.

The following constraints must be met for a valid configuration.

- Down ports must equal up ports: leaf up ports (`leaf_switches_per_subtree` · `up_ports_per_leaf_switch`) must equal aggregation down ports (`agg_switches_per_subtree` · `down_ports_per_agg_switch`), and total aggregation

up ports (`up_ports_per_agg_switch` · `agg_switches_per_subtree` · `num_agg_subtrees`) must equal total core down ports (`num_core_switches` · `down_ports_per_core_switch`).

- Need enough down ports – each switch must have at least one link into each "unit" (subtree or switch, depending on level) below it: `down_ports_per_core_switch` must be ≥ `num_agg_subtrees`, and `down_ports_per_agg_switch` must be ≥ `leaf_switches_per_subtree`.
- Need enough up ports – each "unit" (subtree or switch) must have at least one link into each switch above it: `up_ports_per_agg_switch` · `agg_switches_per_subtree` must be ≥ `num_core_switches`, and `up_ports_per_leaf_switch` must be ≥ `agg_switches_per_subtree`.
- Connections need to be regular:
  - `down_ports_per_core_switch` mod `num_agg_subtrees` must equal zero
  - `down_ports_per_agg_switch` mod `leaf_switches_per_subtree` must equal zero
  - `up_ports_per_leaf_switch` mod `agg_switches_per_subtree` must equal zero

4.3.1 Switch Crossbar Bandwidth Scaling

Allowing non-uniform switches in the topology implies that switch crossbar bandwidth should be non-uniform as well. By default, SST assumes `switch.xbar.bandwidth` specifies the bandwidth for the switch type with the lowest port count. The crossbar bandwidth is scaled by the total number of ports for all other switch types. Input keywords are provided to override this default behavior. For the tapered-bandwidth example above, uniform switch bandwidth can be maintained by setting all bandwidth scaling to 1.0:

```
topology.leaf_bandwidth_multiplier = 1.0
topology.agg_bandwidth_multiplier = 1.0
topology.core_bandwidth_multiplier = 1.0
```

4.3.2 Routing

The fat-tree topology should be used in conjunction with `router = fat_tree`, which will maximize the utilization of path diversity. There is a `fat_tree_minimal` router which will use the lowest numbered valid port for any destination; this will result in poor network performance and is primarily useful for testing and perhaps experiments where network contention is desired.

4.4 Cascade

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. A variation on the dragonfly is the cascade topology implemented by Cray on their Aries interconnects. A cascade is sometimes viewed as a generalization of flattened 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 cascade 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 cascade is usually understood through three parameters:

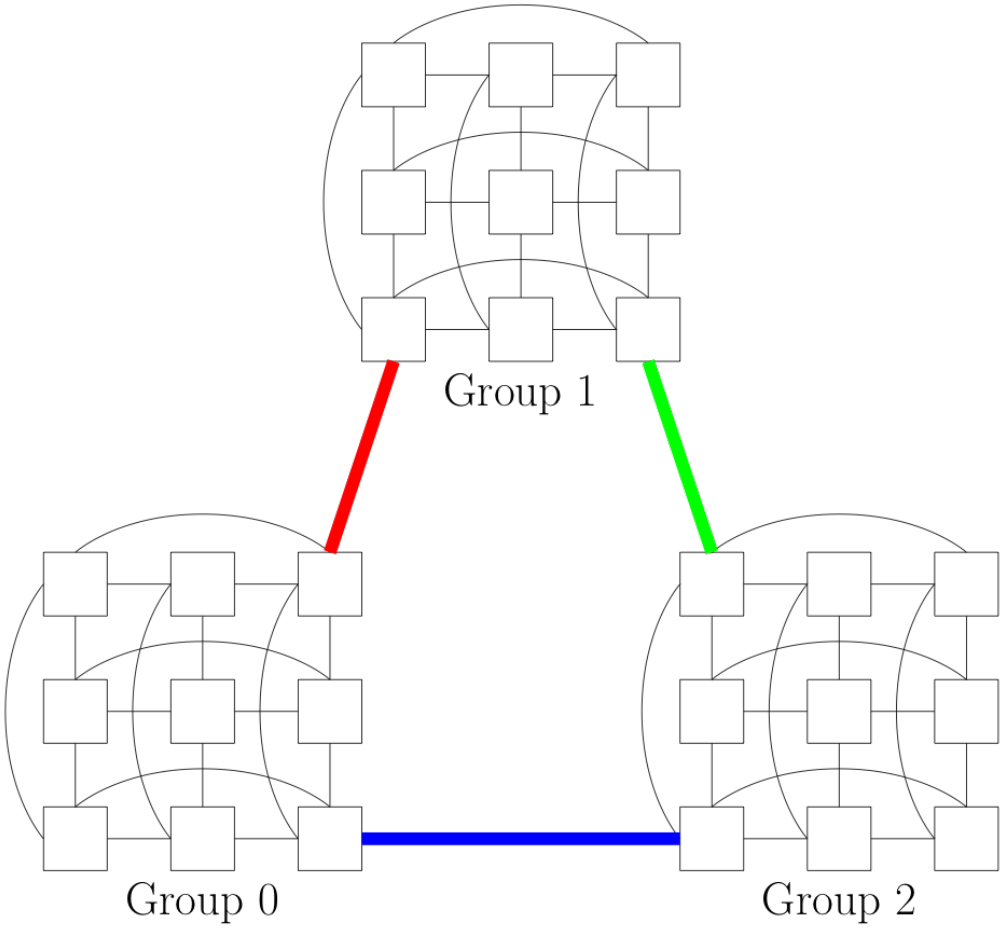


Figure 4.7: Schematic of cascade with three groups showing hypercube intragroup links and high bandwidth intergroup global links

- 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 cascade should not increase the size of a group, only the number of groups.

4.4.1 Allocation and indexing

The cascade 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 = cascade
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
```

4.4.2 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. cascade 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, cascade 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 **??**). 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

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 **??** 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

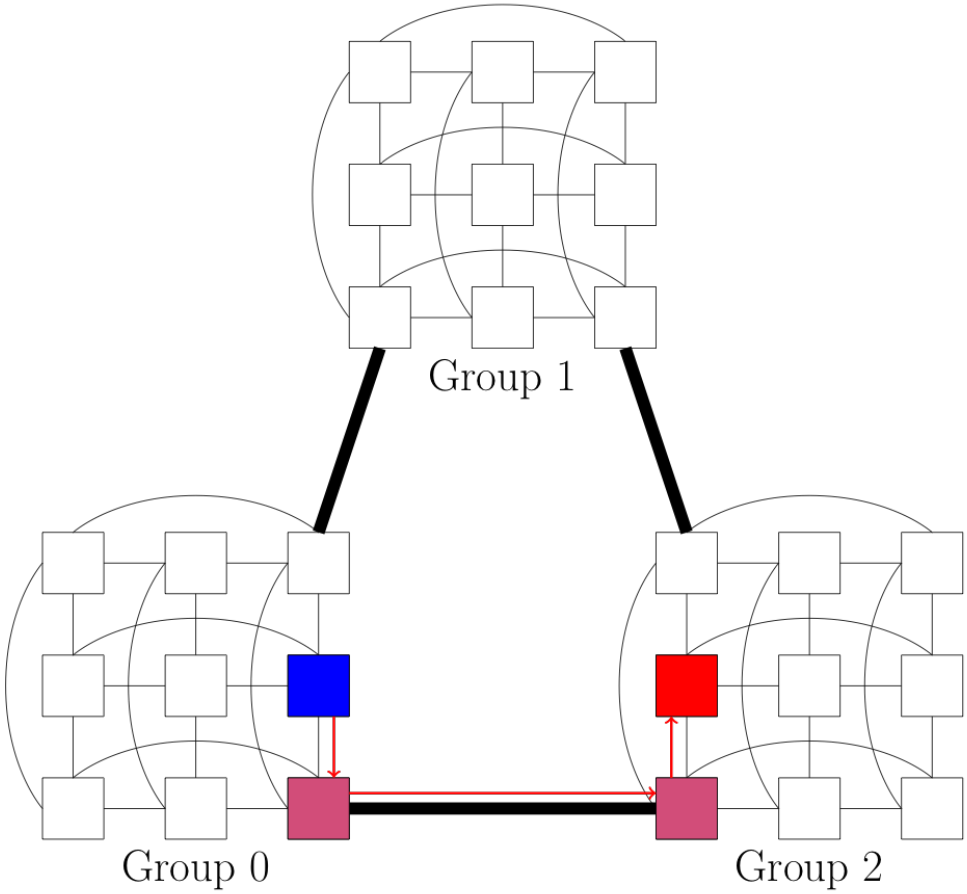


Figure 4.8: Schematic of cascade showing minimal route. Traveling between groups requires routing to the correct global link, hopping the global link, then routing within a group to the correct final node.

```
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:

- 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.

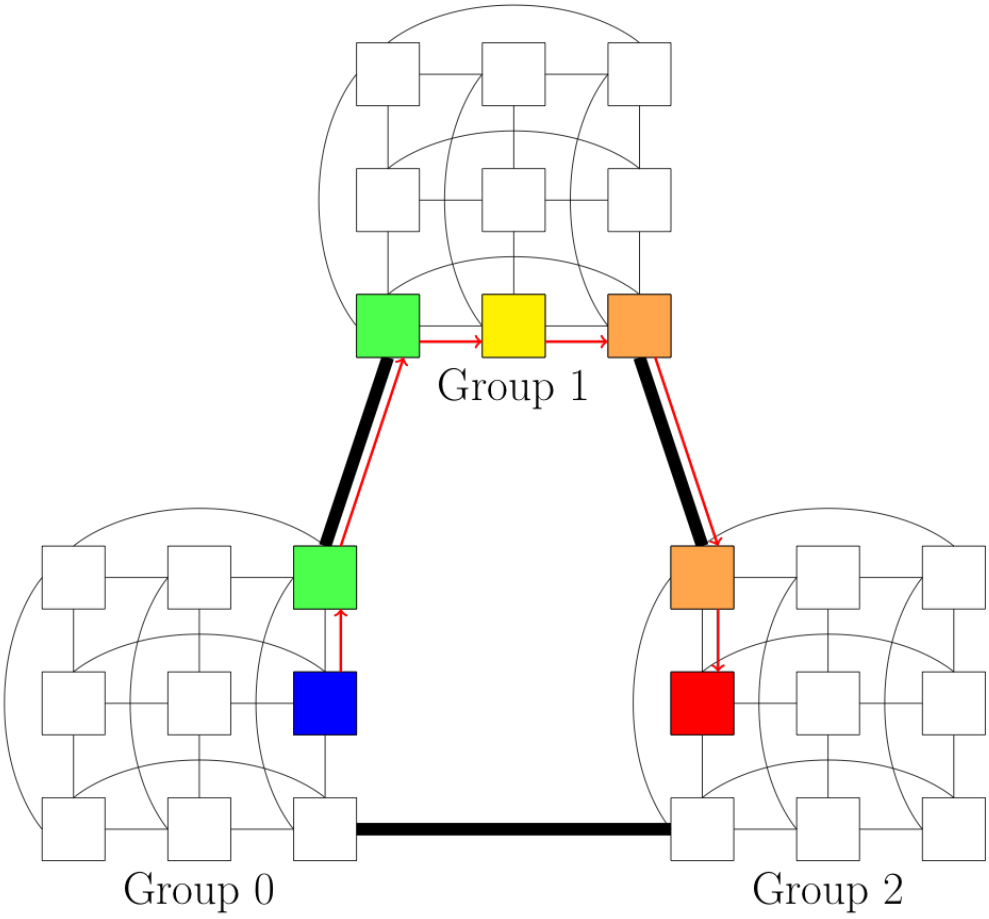


Figure 4.9: Schematic of cascade showing Valiant route. Traveling between groups requires routing to a random intermediate node, then routing minimally to the final destination.

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

External Applications and Skeletonization

5.1 Basic Application porting

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

- Symbol Interception: Rather than linking to MPI, pThreads, or other parallel libraries (or even calling `hostname`), these functions must be redirected to SST/macro rather than calling the native libraries on the machine running the simulator. 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. 1K 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 not its own data segment (global variables). Virtual processes share the same address space and the same global variables. A Beta version of the auto-skeletonizing clang-based SST compiler is available with the 7.X releases. If the Beta is not stable with your application, manual refactoring may be necessary if you have global variables.

Now in Beta, another possible feature is available:

- Memoization hooks: Rather than building an app for simulation, build an app with special profiling hooks. There is no skeletonization or redirected linkage, but the runtime or performance counters obtained by running should be used to build models for simulation.

There are generally 3 modes of using an application common with SST/macro:

- Simulation: As lightweight as possible without sacrificing accuracy. Intended to be used for performance estimation at large scales.

- Emulation/Virtualization: Run a parallel or distributed application within a single simulator thread/process - primarily useful for debugging.
- Memoization: Run a full application within the simulator, collecting performance counters or timers on critical sections

	Simulator hooks	Symbol interception	Skeletonization	Process encapsulation
Simulation	Yes	Yes	Yes	Yes
Virtualization	Yes	Yes	No	Yes
Memoization	Yes	No	No	No

5.1.1 Loading external skeletons with the standalone core

You should always write your own skeleton applications in an external folder rather then integrating directly into the `sstmac` executable. Existing make systems can be used unmodified. Rather than producing an executable, though, SST/macro produces a shared library. These shared libraries are then imported into the main `sstmac` executable.

If you follow the example in the `skeletons/sendrecv` folder, the Makefile shows how to generate an importable skeleton. If you are using `sst++`, it will automatically convert executables into loadable libraries. If your application is named `runapp`, you would run it with `sstmac`:

```
./runapp -f parameters.ini --exe=./runapp
```

directing to load the library as a skeleton executable.

5.2 Auto-skeletonization with Clang

The build of the Clang toolchain is described in Section 2.3. This enables a source-to-source translation capability in the `sst++` compiler that can auto-skeletonize computation and fix global variable references. Some of this can be accomplished automatically (global variables), but most of it (removing computation and memory allocations) must occur through pragmas. A good example of skeletonization can be found in the lulesh2.0.3 example in the skeletons folder. Most of the available SST pragmas are used there. Pragmas are preferred since they allow switching easily back and forth between skeleton and full applications. This allows much easier validation of the simulation. The section here briefly introduces the SST pragma language. A complete tutorial on all available pragmas is given in Chapter 6.

5.2.1 Redirecting Main

Your application's `main` has to have its symbols changed. The simulator itself takes over `main`. SST/macro therefore has to capture the function pointer in your code and associate it with a string name for the input file. This is automatically accomplished by defining the macro `sstmac_app_name` either in your code or through a `-D=` build flag to the name of your application (unquoted!). The value of the macro will become the string name used for launching the application via `node.app1.name=X`. Even without Clang, this works for C++. For C, Clang source-to-source is required.

5.2.2 Memory Allocations

To deactivate memory allocations in C code that uses `malloc`, use:

```
#pragma sst malloc
  void* ptr = malloc(...)
```

prior to any memory allocations that should be deactivated during skeleton runs, but active during real runs.
Similarly, for C++ we have

```
#pragma sst new
  int* ptr = new int[...]
```

5.2.3 Computation

In general, the SST compiler captures all `#pragma omp parallel` statements. It then analyzes the for-loop or code block and attempts to derive a computational model for it. The computational models are quite simple (skeleton apps!), based simply on the number of flops executed and the number of bytes read (written) from memory. Consider the example:

```
double A[N], B[N];
#pragma omp parallel for
for (int i=0; i < N; ++i){
  A[i] = alpha*A[i] + B[i];
}
```

The SST compiler deduces $16N$ bytes read, $8N$ bytes written, and $16N$ flops (or $8N$ if fused-multiplies are enabled). Based on processor speed and memory speed, it then estimates how long the kernel will take without actually executing the loop. If not wanting to use OpenMP in the code, `#pragma sst compute` can be used instead of `#pragma omp parallel`.

5.2.4 Special Pragmas

Many special cases can arise that break skeletonization. This is often not a limit of the SST compiler, but rather a fundemental limitation in the static analysis of the code. This most often arises due to nested loops. Consider the example:

```
#pragma omp parallel for
for (int i=0; i < N; ++i){
  int nElems = nElemLookup[i];
  for (int e=0; e < nElems; ++e){
  }
}
```

Auto-skeletonization will fail. The skeletonization converts the outer loop into a single call to an SST compute model. However, the inner loop can vary depending on the index. This data-dependency breaks the static analysis. To fix this, a hint must be given to SST as to what the "average" inner loop size is. For example, it may loops nodes in a mesh. In this case, it may almost always be 8.

```
#pragma omp parallel for
for (int i=0; i < N; ++i){
  int nElems = nElemLookup[i];
  #sst replace nElems 8
  for (int e=0; e < nElems; ++e){
  }
}
```

This hint allows SST to skeletonize the inner loop and "guess" at the data dependency.

5.2.5 Skeletonization Issues

Skeletonization challenges fall 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.

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 number.
- *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, how many particles migrate and how far depends on the actual computed forces. However, in the skeleton, we are not actually computing the forces - only estimated how long the force computation took. 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 again depends on making approximations or probability estimates of where and when box refinement occurs without actually computing everything.

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.
- *Synthetic* - It may be possible to replace communication with randomly-generated data and decisions, which emulate how the original application worked. This occurs in the CoMD skeleton.
- *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. This occurs in the `boxml` aplications.

5.3 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, you should use the `sst++` clang-based compiler wrappers to auto-refactor your code (Section 5.2). This feature is current labeled Beta, but is stable for numerous tests and will be fully supported for release 7.1.

Chapter 6

Clang Source-to-Source Auto-Skeletonization via Pragmas

There are three main examples of auto-skeletonization with pragmas in the SST/macro source code in the `skeletons` directory. These applications are Lulesh, HPCG, and CoMD. The auto-skeletonizing compiler is designed to do three main things:

- Redirect global variable accesses to thread-specific values
- Turn off large memory allocations that would prevent scalable simulation
- Estimate time of compute-intensive kernels instead of executing them

6.1 Pragma Overview

6.1.1 Compiler workflow

The source-to-source compiler operates on a pre-processed source file. The source code transformation generates a temporary source file. This temporary source file is then compiled into the target object file. Global variables require static registration of C++ variables. Here another temporary C++ source file (even if the original file is C) is generated that has all static global variable registrations. The corresponding object file is merged with the original object file, creating a complete SST/macro object file with the transformed code and C++ static registrations. This workflow is shown in Figure 6.1.

6.1.2 Compiler Environment Variables

SSTMAC_SRC2SRC: Default 1

If set to zero, deactivates the source-to-source transformation. The compiler wrapper will then compile the code into the simulator, but will not redirect any global variable accesses or perform any skeletonization.

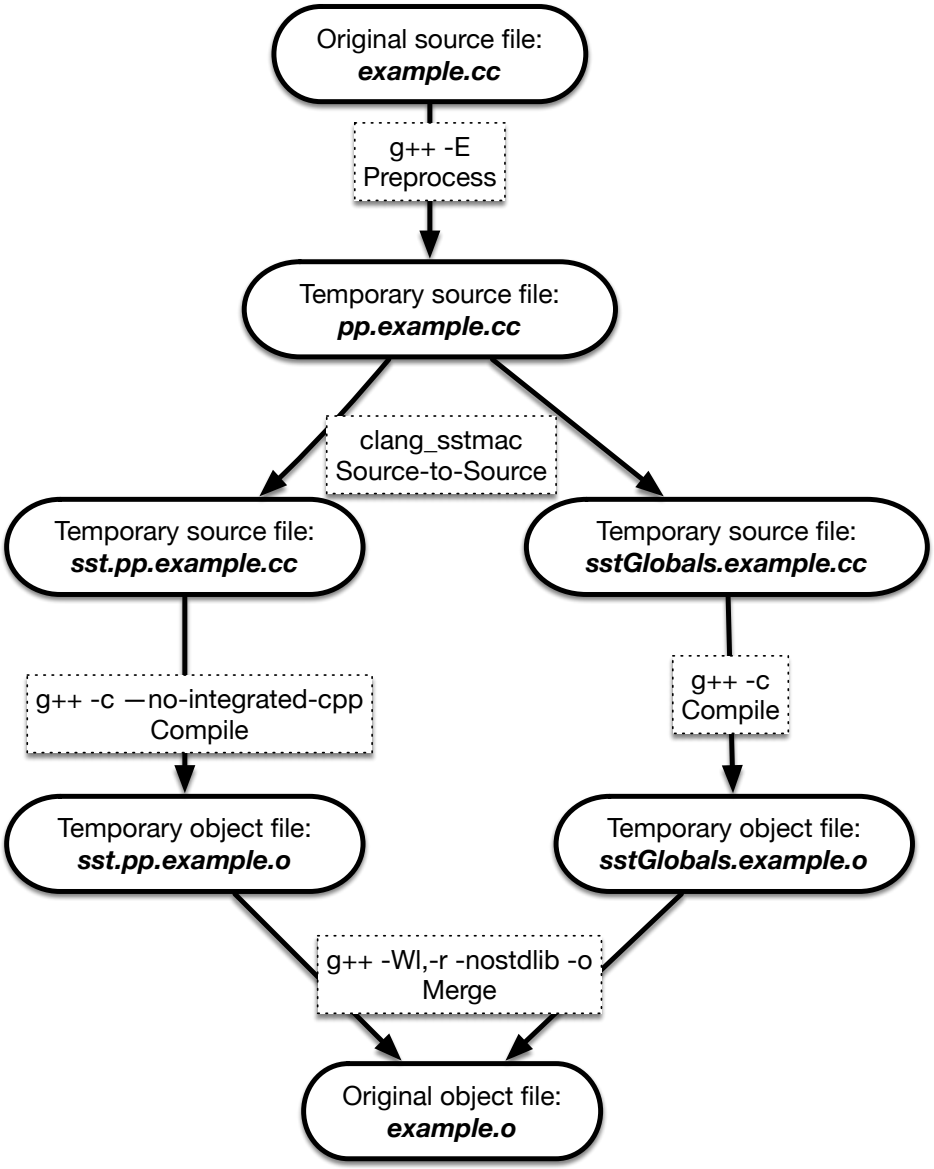


Figure 6.1: Source-to-source transformation workflow for SST compiler. For C source files, g++ can be swapped with gcc. The choice of underlying compiler is actually arbitrary and can be clang, gcc, icc, etc.

SSTMAC_SKELETONIZE: Default 0

If set to zero, deactivates skeletonization. This does not deactivate global variable redirection. Thus, with `SSTMAC_SRC2SRC=1` and `SSTMAC_SKELETONIZE=0`, SST/macro will act as an MPI emulator executing a full code but with global variables refactored

to maintain correctness.

SSTMAC_MEMOIZE: Default 0

If set to nonzero, activates memoization hooks. This deactivates all global variable refactoring, all symbol interception, and all skeletonization.

SSTMAC_HEADERS: No default

The compiler wrapper will only redirect global variables that it knows should definitely be modified. All global variables found in source files will be redirected. `extern` global variables found in header files are more difficult. Certain system global variables like `stderr` should not be modified and so are left as global variable constants. By default, global variables in a header file are NOT redirected unless explicitly specified in a header configuration file. The variable `SSTMAC_HEADERS` should give the full path of a file containing the list of header files. Header file paths in the file should be one per line and should be the full path, not a relative path.

SSTMAC_DELETE_TEMPS: Default 1

If non-zero, the compiler cleans up all temporary files. If you wish to keep temporary files to view them for debugging, set to zero. All temporary, intermediate source files will otherwise be deleted at the end of compilation.

6.2 Basic Replacement Pragmas

When skeletonization is active (see `SSTMAC_SKELETONIZE`), these pragmas will cause replacements in the original source code. Pragmas appy to the next statement in the source code. For compound statements such as a for-loop with a multi-statement body, the pragma applies to the entire for-loop.

6.2.1 pragma sst delete: no arguments

This deletes the next statement from the source code. If the statement declares a variable that is used later in the code, this will cause a compile error. Consider an example from the Lulesh source code.

```
1 #pragma sst delete
2     testnorms_data.values[i] = normr/normr0;
```

In the skeleton, the residual is not actually computed and the `testnorms_data` array is not actually allocated. Thus this statement should be deleted and not actually executed in the skeleton.

6.2.2 pragma sst replace [to_replace:string] [new_text:C++ expression]

This applies a string replace to a variable or function call in the next statement. Consider an example from Lulesh.

```
1 #pragma sst replace volo 1
2    deltatime() = (Real_t(.5)*cbrt(volo(0)))/sqrt(Real_t(2.0)*einit);
```

The function call `volo(0)` is not valid in the skeleton since volumes are not actually computed. Here we simply estimate that all cells have unit volume replacing `volo(0)` with `1`.

6.2.3 pragma sst init [new_value:string]

This pragma can only apply to a binary equals operator assigning a value. The pragma changes the right-hand side to use the given new value. For example, in Lulesh:

```
#pragma sst init nullptr
  destAddr = &domain.commDataSend[pmsg * maxPlaneComm] ;
```

The send buffer `domain.commDataSend` is not allocated in the skeleton and thus is not valid to use. The pragma causes the skeleton to simply set `destAddr` to `nullptr`.

6.2.4 pragma sst return [new_value:C++ expression]

Pragma is equivalent to `pragma sst init`. This replaces the target of a return statement with the given expression. This produces a compiler error if applied to anything but a return statement.

6.2.5 pragma sst keep

During the skeletonization process, some transformations occur automatically even without pragmas. For example, all MPI calls have input buffers converted to null pointers to indicate a simulated MPI call. If the MPI call should be emulated with real payloads, the MPI call must be explicitly marked with `pragma keep`. An example can be found in the HPCG skeleton:

```
#pragma sst keep
  MPI_Allreduce(&localNumberOfNonzeros, &totalNumberOfNonzeros, ...)
```

The actual allreduce operation is carried out, summing the the local number into the total number of nonzeroes.

6.2.6 pragma sst keep_if [condition:C++ bool expression]

More control over whether transformations are skipped is provided by `keep_if`. An example is found in CoMD.

```
#pragma sst keep_if count < 16
   MPI_Allreduce(sendBuf, recvBuf, count, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

Any small allreduce operations are kept. Any allreduce operations larger than a given cutoff are simulated without emulating the actual buffer operations.

6.2.7 pragma sst empty

This pragma is applied to functions. The function prototype is kept, but an empty body is inserted instead of the actual code. This can be useful for deleting large blocks of computation inside a function.

6.2.8 pragma sst branch_predict [condition:C++ expression]

The branch prediction pragma can be applied in two different contexts. We will revisit the pragma in the context of compute skeletonization below. The branch prediction pragma should only be applied to an if-statement. Much like the replace pragmas, it swaps out the given if condition with a new expression.

The branch prediction pragmas become necessary when skeletonizing. Certain values may not be computed or certain variables marked null. If these values are then used in an if predicate, the skeletonizing compiler cannot deduce the correct behavior for the application. When an ambiguous predicate is found, the compiler will usually print a warning and just assume the predicate is false. Predicates are often almost always true or almost always false. Thus most uses of this pragma will simply supply `true` or `false` as the replacement. However, any arbitrary C++ boolean expression can be given as the replacement. The new predicate expression (like the original), must not involve any null variables.

6.3 Memory Allocation Pragmas

6.3.1 pragma sst malloc

Applied to any statement in which the right-hand side is a malloc. This sets the left-hand side to `NULL`. This is critical for turning off large memory allocations on data structures not required for control-flow.

6.3.2 pragma sst new

Applied to any statement in which the right-hand side a C++ operator new. This sets the left-hand side to `nullptr`. This is critical for turning off large memory allocations on data structures not required for control-flow.

6.4 Data-Driven Type Pragmas

6.4.1 pragma sst null_variable

This applies to variable declarations. If pragma is not applied to a declaration, a compiler error is given. A null variable is one in which all operations involving the variable should be deleted. This usually applies to large data arrays that should never be allocated and therefore never dereferenced. An example can be seen in CoMD:

```
1 #pragma sst null_variable
2    int* nAtoms;         //!< total number of atoms in each box
```

The array is not allocated and all statements operating on the array are deleted.

This pragma is much more powerful than simply using `pragma sst new`. `pragma sst new` simply turns off a given memory allocation setting it to a null value. If the array is dereferenced later in code, this causes a segmentation fault. By marking a declaration as null, the compiler can flag where segmentation faults would occur when running the skeleton.

In most cases, all operations involving the null variable are deleted. However, there may be cases where the compiler may decide deleting an operation cannot be done automatically since it may affect control flow, e.g., if the variable is used inside an if-statement. When this occurs, a compiler error is thrown flagging where the ambiguity occurs. Another pragma must then be applied to that statement to tell the compiler how to proceed.

6.4.2 pragma sst null_type [type alias] [list allowed functions]

This applies to C++ class variable declarations. If pragma is not applied to a declaration, a compiler error is given. This essentially works the same way as `null_variable`, but allows certain member functions to be kept instead of deleted. Consider an example from Lulesh:

```
#pragma sst null_type sstmac::vector size resize empty
   std::vector<Real_t> m_x ;  /* coordinates */
```

Here we wish to indicate the vector is "null" and should not actually allocate memory or allow array accesses. However, we still wish to track the vector size and whether it is empty. The first argument to the pragma is a new type name that implements the "alias" functionality. For `std::vector`, SST/macro automatically provides the alias. For illustration, that code is reproduced here:

```
namespace sstmac {
class vector {
 public:
  void resize(unsigned long sz){
    size_ = sz;
  }

  unsigned long size() const {
    return size_;
  }

  template <class... Args>
  void push_back(Args... args){
    ++size_;
  }

  template <class... Args>
  void emplace_back(Args... args){
    ++size_;
  }

  bool empty() const {
    return size_ == 0;
  }

 private:
  unsigned long  size_;
};
}
```

This empty vector class allows the type to track its size, but not actually hold any data. All places in the Lulesh code that an `std::vector` is used are substituted with the new type.

The remaining arguments to the pragma are the list of functions we wish to mark as valid. In this case, even though the alias vector class provides more functions, we only allow `size`, `resize`, and `empty` to be called.

6.5 Compute Pragmas

6.5.1 pragma sst compute and pragma omp parallel

Compute-intensive should not be executed natively. Instead, a compute model should be used to estimate the elapsed time. Compute model replacements are automatically triggered by any OpenMP parallel pragmas. The corresponding block or

for-loop is not executed, instead calling out to a compute model to estimate time. Currently, compute modeling is done via a very basic static analysis. The static analysis attempts to count the number of integer and floating point operations. It also estimates the number of memory reads and writes. These four counters are passed to a coarse-grained processor model for time estimates. For more details, see `sstmac_compute_detailed` in the source code. Numerous examples can be found in the Lulesh, HPCG, and CoMD skeleton applications.

6.5.2 pragma sst loop_count [integer: C++ expression]

If the `sst compute` or `omp parallel` pragma are applied to an outer loop with one or more inner loops, the compute model static analysis might fail. This occurs when the inner loop control flow depends on the actual execution. Any variables declared *inside* the compute block are not valid to use in the compute estimate since they will be skeletonized and deleted. Only variables in scope at the beginning of the outer loop are safe to use in compute modeling.

When the static analysis fails, a corresponding compiler error is thrown. This usually requires giving a loop count hint. Consider the example from HPCG:

```
#pragma omp parallel for
  for (local_int_t i=0; i< localNumberOfRows; i++) {
    int cur_nnz = nonzerosInRow[i];
   #pragma sst loop_count 27
    for (int j=0; j<cur_nnz; j++) mtxIndL[i][j] = mtxIndG[i][j];
  }
```

The static analysis fails on `cur_nnz`. However, that value is almost always 27. Thus we can safely tell the compiler to just assume the loop count is 27.

6.5.3 pragma sst branch_predict [float: C++ expression]

Similar to the way that loop counts can break the static analysis, if statements inside a loop skeletonized with `omp parallel` or `sst compute` can also be problematic. If the predicate depends on a variable declared *inside* the skeletonzied block, the static analysis will break since it cannot predict when and how often to assume true or false. In contrast to the branch prediction pragma previously used, branch prediction pragmas inside a compute block must give a number between 0 and 1. This can either be a literal float or expression that computes a float value. Consider an example from CoMD:

```
#pragma sst branch_predict 0.2
  if(r2 <= rCut2 && r2 > 0.0){
```

Inside the compute block, a compute may or may not occur depending on whether a particle distance is less than a cutoff. Based on the way CoMD constructs unit cells and halo regions, running CoMD shows that about 1 in 5 neighbor interactions are actually below the cutoff. Thus we given the branch prediction the hint 0.2.

6.5.4 pragma sst advance_time [units] [time to advance by]

This pragma advances the simulator time by the specified amounts of time. It can be placed before any statement. The units can be the following: sec, msec, usec or nsec for Seconds, milliseconds, microseconds and nanoseconds respectively.

6.6 Memoization pragmas

6.6.1 Memoization models

To understand the memoization pragmas, we first introduce how models get constructed in the SST/macro runtime. Source-to-source transformations based on the pragmas causes the following hooks to get inserted:

```
int sstmac_startMemoize(const char* token, const char* model);
void sstmac_finish_memoize0(int tag, const char* token);
void sstmac_finish_memoize1(int tag, const char* token, double p1);
void sstmac_finish_memoize2(int tag, const char* token, double p1, double p2);
...
```

A start call begins a memoization region for a specific name. The start function must return an integer tag identifying the memoization instance. This tag gets passed back into the finish function above. This is primarily useful for thread-safe collection, but can be generally more useful. The finish functions take input parameters. Given input parameters x,y causes a function $F(x, y)$ to be fit to the timer or performance counters.

If building a skeleton application that uses memoization data, a different hook gets inserted:

```
void sstmac_compute_memoize0(const char* token);
void sstmac_compute_memoize1(const char* token, double p1);
void sstmac_compute_memoize2(const char* token, double p1, double p2);
...
```

Assuming a model $F(x, y)$ has been fit in a memoization pass, that function is invoked with the given parameters to estimate a time or performance counter.

Memoization models are implemented by inheriting from a standard class

```
struct RegressionModel {
...
virtual double compute(int n_params, const double params[], ImplicitState* state) = 0;
virtual int StartCollection() = 0;
virtual void finishCollection(int n_params, const double params[], ImplicitState* state) = 0;
...
```

A call to `sstmac_finish_memoize2` causes `finishCollection(2,..)` to get invoked on the model. The `states` object is discussed more later in 6.6.3. For now, `compute` only returns a double (total time). Generalized performance models are planned for future versions. Models are registered using the SST/macro factory system. If wanting to add a least-squares model, factory register as:

```
struct least_squares : public regression model {
 FactoryRegister("least_squares", OperatingSystem::RegressionModel, least_squares)
```

6.6.2 pragma sst memoize [skeletonize(...)] [model(...)] [inputs(...)] [name(...)]

- skeletonize: boolean for whether code block should still be executed or remove entirely (default: true)
- model: string name for a type of model (e.g. linear, kmeans) specifying which model to construct and fit (no default)
- inputs: a comma-separated list of C++ expressions that are the numeric inputs
- name: a unique name to use for identifying the memoization region (default: see below)

If the `name` parameter is not given, the file and line number is used for basic expressions while the function name is used if applied to a function. Consider the example:

```
#pragma sst memoize skeletonize(true) model(least_squares) inputs(ncol,nlink,nrow)
void dgemm(int ncol, int nlink, int nrow, double* left, double* right);
```

When running the memoization pass, the memoization hooks get invoked as:

```
int tag = sstmac_startMemoize("dgemm", "least_squares");
dgemm(....);
sstmac_finish_memoize3(tag, "dgemm", ncol, nlink, nrow);
```

With `skeletonize` set to true, the skeleton app would be:

```
sstmac_computeMemoize("dgemm", ncol, nlink, nrow);
```

With skeletonize set to false:

```
sstmac_computeMemoize("dgemm", ncol, nlink, nrow);
dgemm(...);
```

Both the memoization function and the original function would both get invoked.

6.6.3 pragma sst implicit_state X(Y) ...

The implicit state pragma sets certain hardware or software states not captured by the inputs to the memoization pragma. This might involve DVFS states, different runs of a task in which data is "cold" or "hot" in cache, or different types of cores. The implicit state lasts for the scope of the statement:

```
#pragma sst implict_state ...
{
 //all statements here have that state
}

#pragma sst ImplicitState ...
fxn(...) //implicit state lasts the entire function
```

The arguments to the pragma are best understood by example:

```
#pragma sst ImplicitState dvfs(1) cache(hot)
fxn(...)
```

This causes a source code transformation to:

```
sstmac_set_ImplicitState2(dvfs,1,cache,hot);
fxn(...);
sstmac_unset_ImplicitState2(dvfs,cache);
```

For now, the functions take integer arguments (this may get relaxed to arbitrary strings). Thus, e.g. enums must be available or compilation will fail:

```
enum states {
 dvfs=0,
 cache=1
};
enum cache_states {
 cold=0,
 hot=1
};
```

If a `sstmac_finish_memoize` function got invoked, the states could be read. The class `ImplicitState` is a base class only and carries no data by default. Specific memoization models are intended to be used only with known implicit state classes. As such, the memoization model `collect`, etc, functions must dynamic cast to an expected type. A library of standard implicit state implementations is planned for future releases.

Chapter 7

Uncertainty Quantification Methods and Tools

7.1 Overview

The SST/macro UQ workflow generally occurs in two steps:

- Generating and running parameter sweep
- Running calibration and sensitivity analysis

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

UQTk Version 2.0 is released under the GNU Lesser General Public License (LGPL). A tar-ball with the source code, tutorials, examples and documentation can be downloaded from `www.sandia.gov/UQToolkit/uqtk_download.html` UQTk uses a standard CMake build system. Fortran (and libgfortran) is required. The two basic UQ tasks, enabled by UQTk, are *forward UQ* and *inverse UQ* as outlined in the next subsections.

7.2 Parameter Sweep and Data Collection

Performing uncertainty quantification and sensitivity analysis requires sampling across a multi-dimensional parameter space. Here the multi-dimensional input parameters covers all the latency, bandwidth, and size parameters for a given machine configuration. In practical usage, a large parameter sweep must be performed to build a surrogate model. The surrogate model must cover all observables - outputs or statistics of interest from the simulation. That surrogate model is then used in a Markov Chain Monte-Carlo (MCMC) workflow to derive properties such as:

- Output sensitvity to input parameters
- Maximum likelihood values of input parameters
- Posterior distribution showing for each parameter showing uncertainties

7.2.1 Surrogate Construction

SST/macro provides helper scripts in the folder `bin/uq` to generate and collect the data for surrogate construction. The following steps must be performed:

- Create a template file (see `exampleTemplate.ini` for example). Variables such as $link_bw are marked to be replaced.
- Create a parameter definition file (see `exampleParams` example file). Each variable to explore should match the $name in the template file. Each parameter must define the range of values to explore. These define the "prior distribution" of realistic values for each parameter with min and max for the range given.
- Run `generate_quad` installed by UQTk. This will create a file `qdpts.dat` defining the parameter sweep to perform.
- Run `genSweep` using `qdpts.dat` to generate all the input `params_N.ini` files for each point in the sweep.
- Run `genRunScript` to create a script `runAll` for running all the jobs in parallel on a given machine.
- Make sure the `runSubSweep` script is in the same folder as `runAll`. Run the `runAll` script. Wait until a `testN.out` file matching all the .ini files is generated.
- Gather the results. This must be re-written for each study. An example file `gatherResults` shows the basic idea. A final file, e.g. `results.out`, must be created with one line for each .ini file. Each line must contain all the output observables corresponding to the run.

The files `qdpts.dat` and `results.out` are enough to build the surrogate.

7.2.2 Surrogate Validation

The surrogate is a polynomial fit to the actual observed values. To test how well the surrogate is able to reproduce the actual simulation, a random set of parameter points should be run through the simulator and compared to the surrogate estimates. SST/macro provides a helper script `genRandom` to create a `samples.dat` file. The `samples.dat` file is equivalent to `qdpts.dat` in the surrogate construction phase. Once `samples.dat` is generated, all the same steps should be followed as in Section 7.2.1.

7.2.3 Experimental Comparison

The simulator is trying to reproduce experimental values collected from an actual machine. These experimental values are the "nominally correct" values. The parameter calibration and sensitivity analysis, must compare to the nominally correct values. Usually these will come from a test bed or existing system, but may come from, e.g. higher-accuracy simulations. A file `experiments.out` should be generated that matches the format of `results.out` (one line per experimental trial, all output observables on a given line). While only a single line of data is minimally required, having multiple trials will improve the UQ workflow by including experimental noise into the uncertainty quantification.

Once you have all three output files (simulation results, random simulation samples, and experimental results), surrogate construction and analysis can take place using UQTk.

7.2.4 Initial Sanity Checks

To sanity check the match between the simulation parameter sweep and the experimental values, SST/macro provides a helper script `validatePoints` that creates a PDF plot of all the data. This is illustrated in Figure 7.1 for an MPI ping-pong benchmark. Here the entire range of the parameter sweep is shown in red lines for the simulator values. The range of experimental values for each output is also shown. In this case, the parameter sweep covers all the experimental outputs suggesting that we can reproduce the experiments.

7.3 Advanced Usage: Running Surrogate Construction and Sensitivity Analysis By Yourself

Running the UQ analysis is involved. Help is available by contacting us at `sst-macro-help@sandia.gov`. Documentation pending...

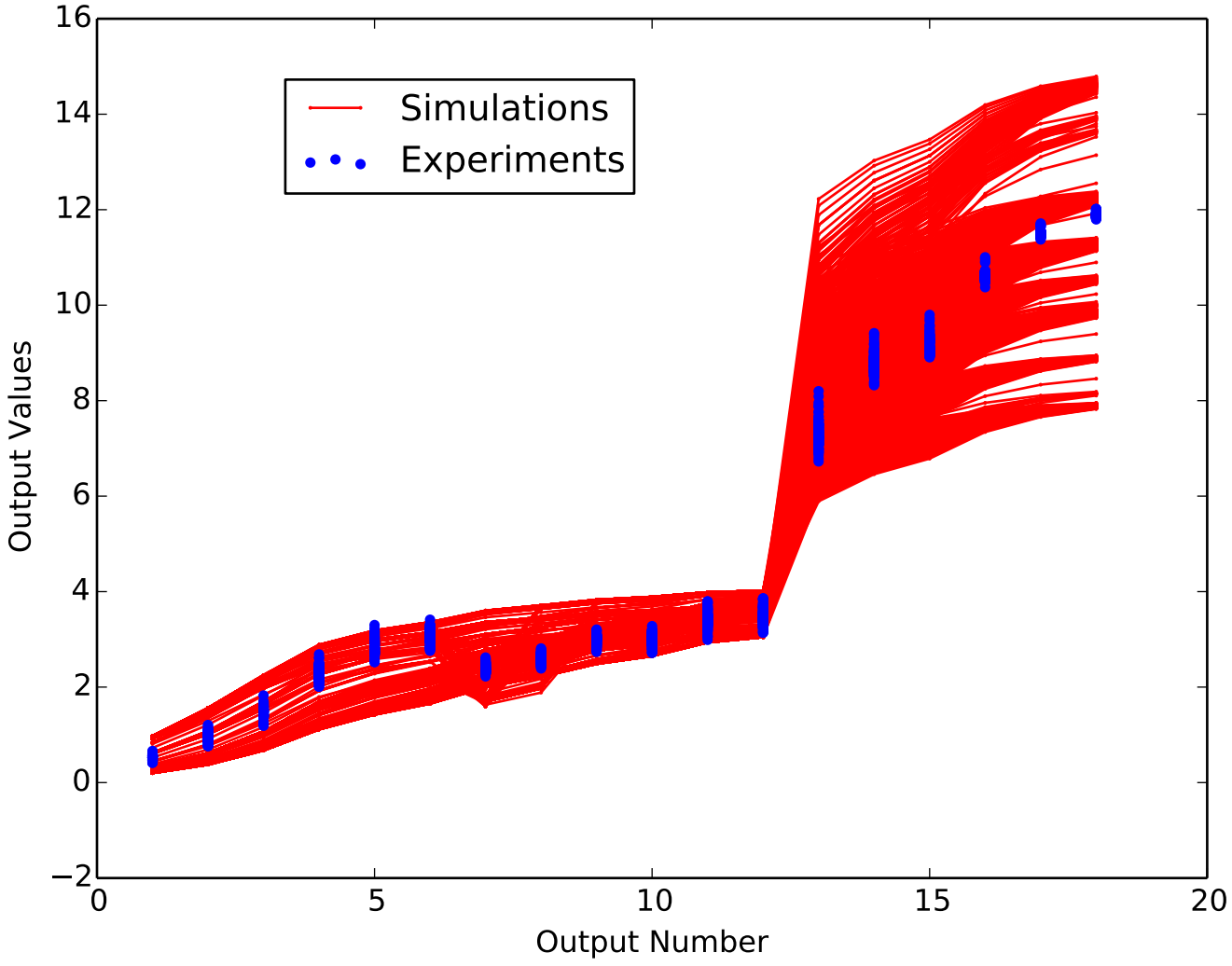


Figure 7.1: Sanity check plot showing the parameter sweep data (red lines) against the experimental outputs (blue dots) for an MPI ping-pong benchmark

Chapter 8

Issues and Limitations

8.1 Polling in applications

Use of probe non-blocking functions in a loop, such as:

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

creates problems for the simulation. Virtual 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.

8.2 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 9

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

9.1 Global namespace

sst_nthread (int)	1	Positive int	Only relevant for multi-threading. Specifying more threads than cores can lead to deadlock.
serialization_buffer_size (byte length)	16 KB		Size to allocate for buffering point-to-point sends in parallel. This should set be large enough to handle serialization of all messages in a given time window, but not so large that significant space is wasted.
backup_buffer_size (byte length)	1 MB		Size to allocate for special overflow buffers when the standard buffer is overrun. This is the base size and continues to grow if buffers overflow again in a time window. This should be large enough so that buffers do not continuously overflow, but not so large that memory gets filled up.
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.

9.2 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
auto (bool)	false	Whether to auto-generate the topology based on the application size.	
name (string)	No default	torus, cascade, dragonfly, fat_tree, 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 cascase ir 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, cascadem, dragonfly, torus) this specifies a bandwidth (redundancy) multiplier for network links in each dimension.

9.3 Namespace "node"

name (string)	simple	simple	The type of node model (level of detail) for node-level operations

9.3.1 Namespace "node.nic"

name (string)	No default	pisces, logp, sculpin	The type of NIC model (level of detail) for modeling injection of messages (flows) to/from the network.
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.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`.

Namespace "node.nic.injection"

9.3.2 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.

9.3.3 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)	64 KB		The size of user-space thread stack to allocate for each virtual application
stack_chunk_size (byte length)	1 MB		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.

9.3.4 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.

9.4 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
otf2_dir_basename (time)	empty string		Enables OTF2 and combines this parameter with a timestamp to name the archive

9.4.1 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

9.5 Namespace "switch"

name (string)	No default	logp, pisces, sculpin	The type of switch model (level of detail) for modeling network traffic.
mtu (byte length)	1024B		The packet size. All messages (flows) will be broken into units of this size.

9.5.1 Namespace "switch.router"

name (string)	No default	minimal, valiant, ugal, dragonfly_minimal, 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.

9.5.2 Namespace “switch.xbar"

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		The latency to traverse the component
bandwidth	No default		The bandwidth of the arbitrator
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.

9.5.3 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		The latency to traverse the component
bandwidth	No default		The bandwidth of the arbitrator
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.

9.6 Namespace “appN”

This is a series of namespaces `app1`, `app2`, and so on for each of the launched applications. These should be contained within the `node` namespace.

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
host_compute_timer (bool)	False		Use the compute time on the host to estimate compute delays

otf2_metafile (string)	No default	string	The root file of an OTF2 trace.
otf2_timescale (double)	1.0	Positive float	If running OTF2 traces, scale compute times by the given value. Values less than 1.0 speed up computation. Values greater than 1.0 slow down computation.
otf2_print_mpi_calls (bool)	false		Print MPI calls found in the OTF2 trace
otf2_print_trace_events (bool)	false		Debugging flag that printsindividual trace events (which includes details such as when an MPI call begins, ends, and when a collective begins and ends
otf2_print_time_deltas (bool)	false		Debugging flag that prints compute delays injected by the simulator
otf2_warn_unknown_callback (bool)	false		Debugging flag the prints unknown callbacks