Sandia National Laboratories

Exceptional service in the national interest

Simulating System Level Interconnects with SST

HPCA 2024

Presented by: Scott Hemmert, Sandia National Laboratories

Content by: SST Team

Edinburgh, United Kingdom

Sunday, March 3

SAND2024-02470C

Welcome!

Part 1: Introduction to SST	*8:00 – 10:00*
Break	10:00 – 10:20
Part 2: Node-Level Simulation	10:20 – 11:00
Part 3: System-Scale Simulation	**11:00 – 12:20**

- SST Overview
- Hands on: Basic Simulation in SST
- A Tour of SST Elements

Learning Objectives

By the end of this section of the tutorial, you will:

- Understand the architecture of the Merlin models
- Know how to use the ember endpoint models with Merlin
- Know how to add new functionality into Merlin through an external element library

High-level View

Merlin: Network Simulator

Low-level, flexible networking components that can be used to simulate high-speed networks (machine level) or on-chip networks

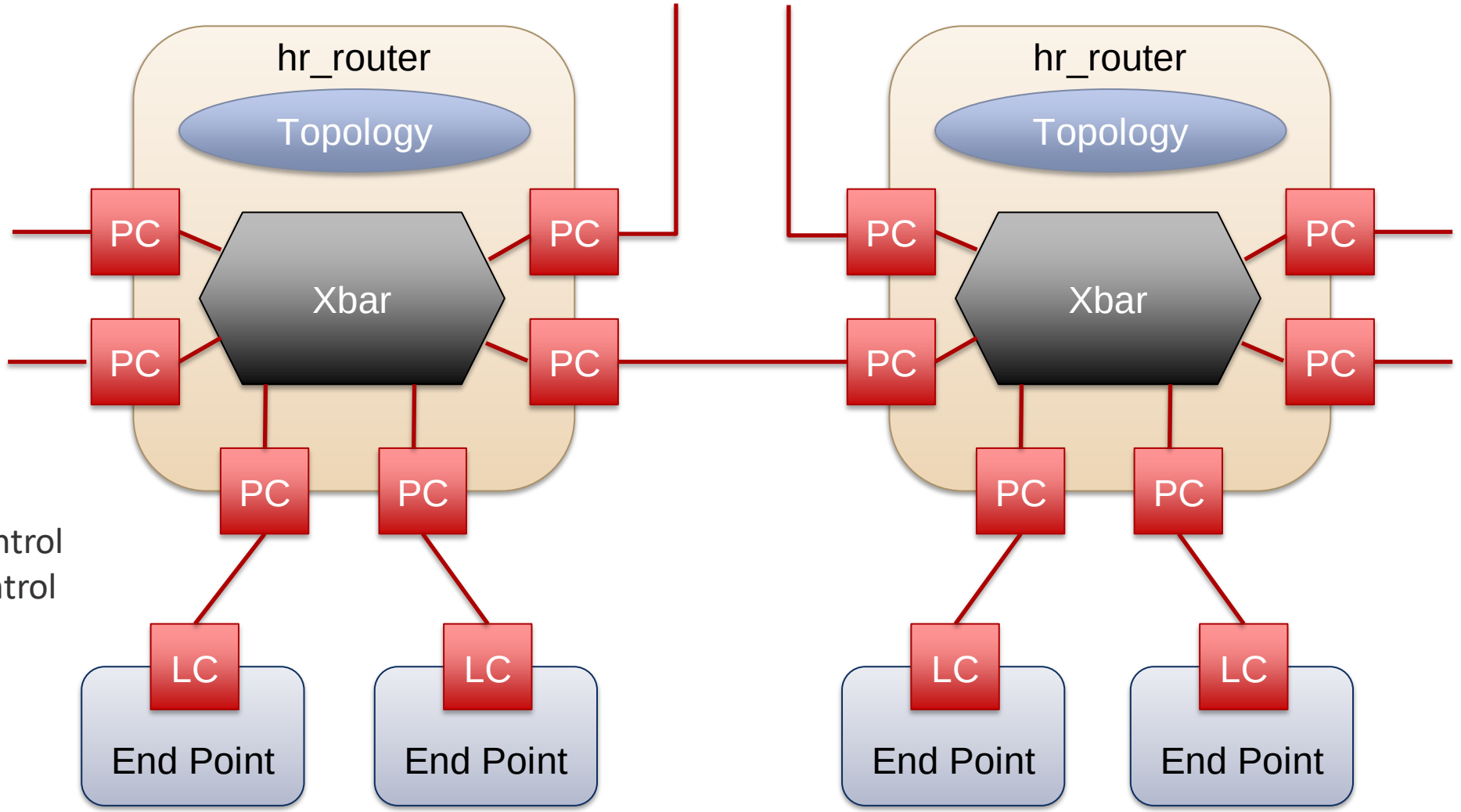
Capabilities

- High radix router model (hr_router)
- Topologies – mesh, n-dim tori, fat-tree, dragonfly, hyperx

Many ways to drive a network

- Simple traffic generation models
- *MemHierarchy*
- Lightweight network endpoint models (*Ember*)
- Or, make your own

Merlin High Level Overview



PC = PortControl
LC = LinkControl

General Parameters

General

- link_bw – bandwidth of the network links (in B/s or b/s)

hr_router

- xbar_bw – per port bandwidth into router crossbar (in B/s or b/s)
- flit_size – size of a flit (in B or b)
- input_latency – input latency of router (in s)
- output_latency – output latency of router (in s)
- xbar_arb – crossbar arbitration unit to be used

hr_router/LinkControl

- input_buf_size – size of input buffer for router/NIC (in B or b)
- output_buf_size – size of output buffer for router/NIC (in B or b)

LinkControl – Endpoint Facing

Inherits from SST::Interfaces::SimpleNetwork

```
bool send(SST::Interfaces::SimpleNetwork::Request* req, int vn);
bool spaceToSend(int vn, int flits);

SST::Interfaces::SimpleNetwork::Request* recv(int vn);
bool requestToReceive( int vn ) { return ! input_buf[vn].empty(); }

void sendUntimedData(SST::Interfaces::SimpleNetwork::Request* ev);
SST::Interfaces::SimpleNetwork::Request* recvUntimedData();

void setNotifyOnReceive(HandlerBase* functor) { receiveFunctor = functor; }
void setNotifyOnSend(HandlerBase* functor) { sendFunctor = functor; }

bool isNetworkInitialized() const { return network_initialized; }
nid_t getEndpointID() const { return id; }
const UnitAlgebra& getLinkBW() const { return link_bw; }
```

SimpleNetwork::Request

```
class Request : public SST::Core::Serialization::serializable {
public:
    nid_t  dest;         /*!< Node ID of destination */
    nid_t  src;          /*!< Node ID of source */
    int    vn;           /*!< Virtual network of packet */
    size_t size_in_bits;  /*!< Size of packet in bits */
    bool   head;         /*!< True if this is the head of a stream */
    bool   tail;         /*!< True if this is the tail of a steram */

private:
    Event* payload;      /*!< Payload of the request */
public:
    inline void givePayload(Event *event);
    inline Event* takePayload();
    inline Event* inspectPayload();

protected:
    TraceType trace;
    int traceID;
```

LinkControl-PortControl Interactions

LinkControl and PortControl share data and negotiate various parameters during the init() phase

- Each LC/PC pair will negotiate link bandwidth. It is set to the minimum of the two set bandwidths
- PortControl will notify the LinkControl of the network ID used to address it
- PortControl will report FLIT size to the LinkControl
- LinkControl notifies PortControl of the desired Virtual Networks to be used
  - This is a deprecated feature. Number of VNs is now set directly through the router
- The LinkControl and PortControl objects manage send credits

PortControl/LinkControl Statistics

packet_latency (LC only)
- For each packet, adds latency to statistics object (side effect is that it counts the number of packets received at an endpoint)

send_bit_count
- For each packet, adds the size in bits to the statistics object (side effect is that it counts number of packets sent on a port)

output_port_stalls
- For each interval where data is present, but can't be sent due to lack of send credits, add the time stalled to statistics object

idle_time (PC only for now)
- For each interval where no data is present to be sent, add total idle time to statistics object

ReorderLinkControl

Inherits from SST::Interfaces::SimpleNetwork

Contains a SimpleNetwork interface object to talk to the “physical” layer (this is typically just a LinkControl).

Puts a sequence number on each packet and reorders packets on the receive-side before giving them to endpoint

- Allows endpoint models that can’t handle out of order receipt of packets to use network models that don’t guarantee ordering

Currently assumes “infinite” resources for buffer and reordering

Crossbar Arbitration

xbar_arb_rr
- Round robin allocation – first across ports, then across virtual channels

xbar_arb_lru
- Least recently used – across all port/VC pairs

xbar_arb_age
- Oldest packets get higher priority

xbar_arb_rand
- Random priority assigned to each port/VC pair each arbitration cycle

Topology Module

Router knows nothing about topology/routing without loading a topology module

- Can use static and/or dynamic routing

Available topologies

- SingleRouter
- Mesh
- Torus
- Fattree
- HyperX/Flattened Butterfly
- Dragonfly
- PolarFly/PolarStar

Topology Class – More on this later

```
virtual void route_packet(int port, int vc, internal_router_event* ev) = 0;
virtual internal_router_event* process_input(RtrEvent* ev) = 0;

virtual PortState getPortState(int port) const = 0;
bool isHostPort(int port) const;
virtual std::string getPortLogicalGroup(int port) const;

virtual void routeInitData(int port, internal_router_event* ev, std::vector<int> &outPorts) = 0;
virtual internal_router_event* process_InitData_input(RtrEvent* ev) = 0;

virtual int computeNumVCs(int vns);
virtual int getEndpointID(int port);

virtual void recvTopologyEvent(int port, TopologyEvent* ev);
```

Configuring a Merlin Simulation

Merlin/Ember Python Modules

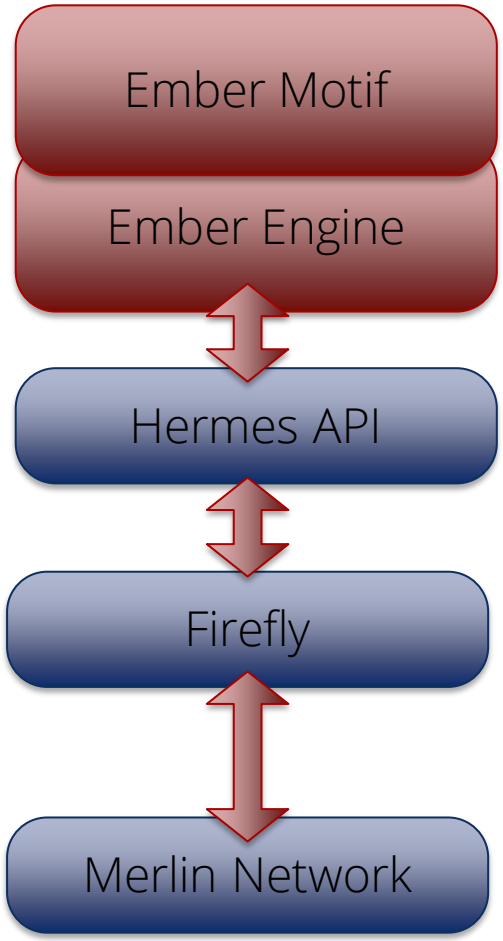
Merlin and Ember provide built-in Python modules to make configuring simulations easier

- Wires up the topology based on user supplied parameters
- Allows simulated jobs (endpoint models) to be easily allocated to the network

The Python modules are built off of five primary base classes:

- System – Overall system with functions to allocated jobs and build the simulation
- Topology – Controls the network topology and parameters
- RouterTemplate – Allows the ability to swap in different router models
  - In practice, there is only one current model: hr_router
- Job – Jobs are mapped to the system using the job allocation functions and contains all the parameters for running the given job on the system
- PlatformDefinition – allows you to capture all or part of the platform parameters and load them into a simulation instead of having to repeat this information in every input file

Ember – Lightweight Network Endpoint



High Level Communication Pattern and Logic
Generates communication events

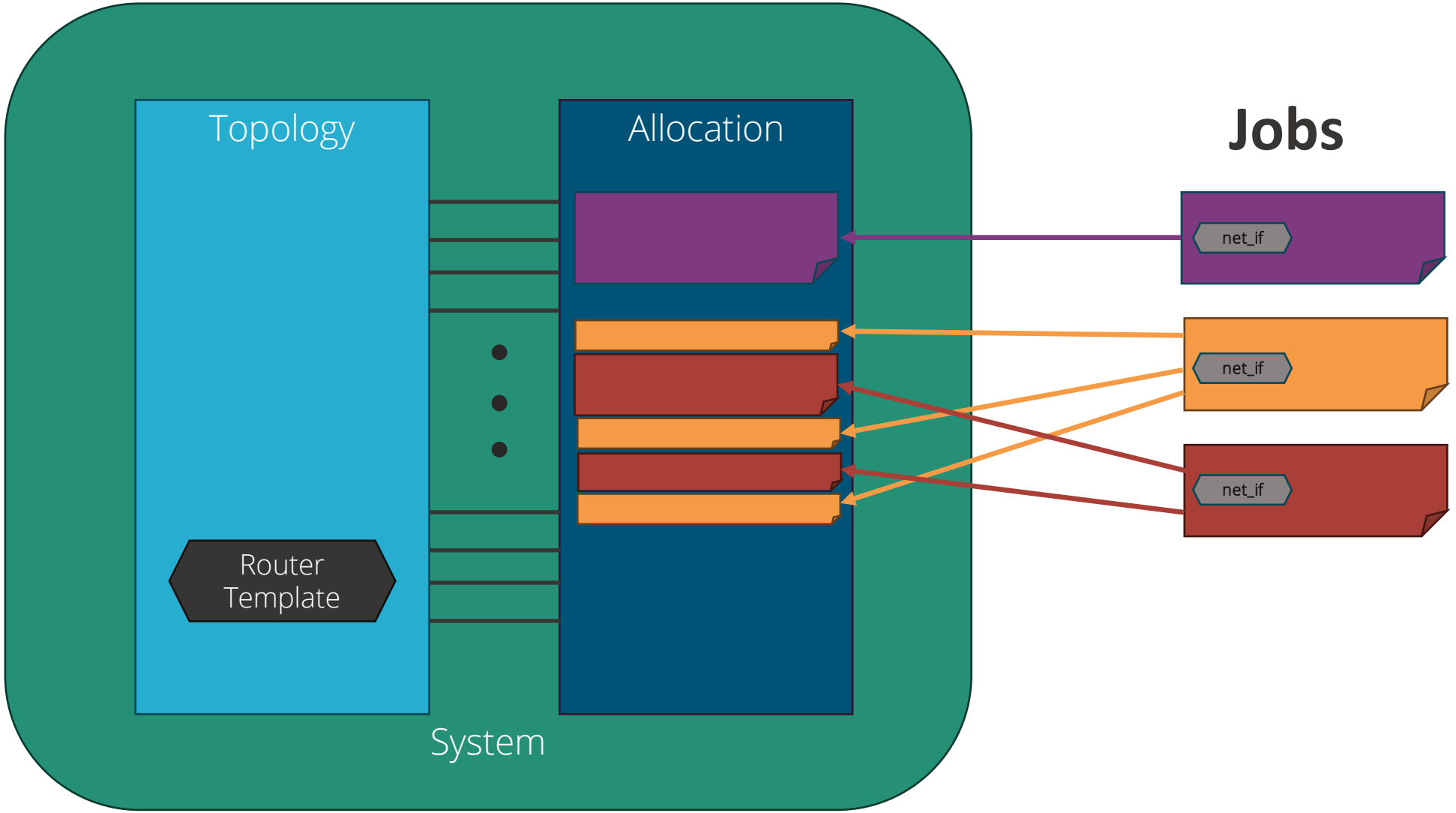
Event to Message Call, Motif Management
Handles the tracking of the motif

Message Passing Semantics
Collectives, Matching, etc.

Packetization and Byte Movement Engine
Generates packets and coordinates with network

Flit Level Movement, Routing, Delivery
Moves flits across network, timing, etc.

Block Diagram of Python Network Configuration Classes



System

After configuring the System, the build() function is called to generate the configuration graph. Configuration options (some of these can be set in a PlatformDefinition file):

- topology: set the topology to use. See next slide for available topologies
- allocation_block_size: sets the number of contiguous endpoints that will be used for allocation. Only currently used when simulating multiNIC nodes
- Allocate jobs to system: jobs are allocated using the allocateNodes() function. The following allocation algorithms can be used, and each job can use a different allocation using the remaining nodes in system:
  - Random – randomly allocate ranks to the available nodes
  - Linear – allocate nodes in order based on number used by topology
  - Random-linear – allocate linearly in a randomly selected subset of nodes
  - Interval – allocate N nodes every M endpoints (this is handy for reserving nodes for things like I/O nodes in the network)
  - Indexed – provide a list of ordered endpoints to allocate the job to (all endpoints in the list must still be unallocated in the system)

Topology

Controls the topology and all the high level network parameters (for example link bandwidth). The build function is called during the System.build() function

Can set the type of router to use by setting MyTopology.router, or can use the default (hr_router)

Supported topologies:

- Dragonfly – uses 1D all-to-all groups
- Hyperx - can be of any dimension, with any number of links per dimension
- Fattree – built using individual routers (no support for consolidated routers)
- Mesh/Torus – can be of any dimension, with any number of links per dimension
- SingleRouter – just a single router (used mostly to mimic crossbars on chip)

Topology: Mesh/Torus

Implements an N-dimensional torus or mesh

shape
- Gives the number of routers in each dimension
- Examples: 16x16x16, 4x4x4x4x4

width
- Gives number of links between routers in each dimension
- Example: 1x1x1, 4x8x4, 3x3x3x3x3

local_ports
- Gives number of hosts connected to each router

```
// Create a Torus topology
topology = topoTorus()

// Set shape to 3D torus with 16 routers per
// dimension
topology.shape = "16x16x16"

// 8 links between routers in each dimension
topology.width = "8x8x8"

// 16 endpoints per router
topology.local_ports = 16
```

Topology: HyperX/Flattened Butterfly

Implements and N-dimensional hyperX/flattened butterfly

shape
- Gives the number of routers in each dimension
- Examples: 16x16x16, 4x4x4x4x4

width
- Gives number of links between routers in each dimension
- Example: 1x1x1, 4x8x4, 3x3x3x3x3

local_ports
- Gives number of hosts connected to each router

```
// Create a hyperX topology
topology = topoHyperX()

// Set shape to 3D torus with 16 routers per
// dimension
topology.shape = "16x8x16"

// Keep same bisection BW for each dimension
// by doubling links in small dimension
topology.width = "1x2x1"

// 16 endpoints per router
topology.local_ports = 16
```

Topology: Dragonfly

Implements a dragonfly topology with fully connected local group

hosts_per_router
- Number of hosts connected to each router

routers_per_group
- Number of routers per local group

intergroup_links
- Number of links between each pair of groups

intragroup_links
- Number of links between each router pair in a group

num_groups
- Total number of groups in the topology

algorithm
- Routing algorithm to use
  - minimal, min-a, valiant, ugal

```
// Create a dragonfly topology
topology = topoDragonFly()

// Set network parameters
topology.hosts_per_router = 16
topology.routers_per_group = 16
topology.intragroup_links = 2
topology.intergroup_links = 8
topology.num_groups = 32

// Set the routing algorithm
topology.algorithm = "ugal"

// Get the total number of nodes
num_nodes = topology.getNumNodes()
```

Total number of hosts is:
- hosts_per_router * routers_per_group * num_groups

Topology: Fattree

Implements an N-level fattree

shape

- Specifies the number of up and down ports at each level of the fattree. Equal up and down links indicates no bandwidth tapering, while unequal numbers indicated tapering. Highest level only specified down links:
  - Specified down,up:down,up:down,up:down
    - Total number of nodes is computed by multiplying all the ”downs” together
  - Examples:
    - 18,18:18,18:36 (largest 3 level fattree using 36 port router with 11,664 hosts)
    - 24,12:18,18:18 (3 level fattree with 50% bandwidth taper out of first level with 7,776 hosts)

These logically match typical fattrees, but are not necessarily physically the same (for example, does not consolidate routers at top level when not all ports are used in each router)

RouterTemplate

Controls which router model is used when the topology is built.

Router parameters are set on the RouterTemplate object. For hr_router, the main parameters are:

- link_bw
- flit_Size
- xbar_bw
- num_vns
- input_latency
- output_latency
- input_buf_size
- output_buf_size

If using the default router, you can access the parameters of the RouterTemplate object as follows:

- MySystem.topology.router.link_bw = “100 Gbps”
- MySystem.topology.router.input_buf_size = “12 kiB”

Job

The Ember Python module uses the Job abstraction to easily configure Ember jobs using the Merlin network models.

To configure a set of motifs using MPI, use EmberMPIJob

- Constructor takes the following parameters:
  - job_id – unique ID to identify the job
  - num_nodes – number of network endpoints to use in the system
  - numCores – number of cores (MPPI ranks) per node to simulate
  - nicsPerNode – number of NICs to simulate per node
    - Each rank will get mapped to only one NIC, and numCores must divide evenly by nicsPerNode
    - The System allocation block size must be a multiple of the nicsPerNode
- Must set the nic parameter to be the network interface corresponding to the router used in topology (for hr_router either LinkControl or ReorderLinkControl)
- Add motifs to the job using the addMotif() function. Examples:
  - myJob.addMotif("Halo3D26 pex=32 pey=32 pez=32 nx=20 ny=20 nz=20 iterations=1")
  - myJob.addMotif("Allreduce")
- There are a large number of other parameters which are best set in a PlatformDefinition file (see next slide)

PlatformDefinition

The various objects in the simulation will “subscribe” to a set of named parameters and class types that can be used to configure the simulation objects, which can set using the PlatformDefinition interface:

- PlatformDefinition.loadPlatformFile(”firefly_platform")
  - Loads a python file that contains platform definitions
  - Platform files can contain more than on platform definition
  - Multiple platform files can be loaded
- PlatformDefinition.setCurrentPlatform(”firefly-platform-base")
  - Set the specified definition as the current platform (can only load one platform at a time)
- PlatformDefinition.compose(platformName, ...)
  - Allows you to compose multiple PlatformDefinitions into a new PlatformDefinition
    - For example, ember parameters and network definition can come from different platform files and be composed together since you can only have one active platform

Platform files can be provided by Sandia, vendors, etc.

Example Configuration using Ember

Example Configuration: merlin-ember-example.py

```
# Import base sst module
import sst

# Import necessary merlin modules
from sst.merlin.base import *
from sst.merlin.endpoint import *
from sst.merlin.interface import *
from sst.merlin.topology import *

from sst.ember import *

# Include the firefly defaults to get default
# parameters for NIC and network stack
PlatformDefinition.setCurrentPlatform(
   "firefly-defaults")
```

```
# Setup the topology
topo = topoDragonFly()
topo.hosts_per_router = 2
topo.routers_per_group = 4
topo.intergroup_links = 2
topo.num_groups = 4
topo.algorithm = ”ugal"]
topo.link_latency = "20ns"

# Set up the routers
router = hr_router()
router.link_bw = "4GB/s"
router.flit_size = "8B"
router.xbar_bw = "6GB/s"
router.input_latency = "20ns"
router.output_latency = "20ns"
router.input_buf_size = "4kB"
router.output_buf_size = "4kB"
router.num_vns = 1
router.xbar_arb = "merlin.xbar_arb_lru"

# Add router template to topology
topo.router = router
```

Example Configuration: merlin-ember-example.py

```
# Set up the network interface
networkif = ReorderLinkControl()
networkif.link_bw = "4GB/s"
networkif.input_buf_size = "4kB"
networkif.output_buf_size = "4kB"

# Create the ember job
ep = EmberMPIJob(0,topo.getNumNodes())
ep.network_interface = networkif
ep.addMotif("Init") # required first motif
ep.addMotif("Allreduce")
ep.addMotif("Fini") # required last motif
ep.nic.nic2host_lat= "100ns"

# Create the system object
system = System()
system.setTopology(topo)

# Allocate the job using linear placement
system.allocateNodes(ep,"linear")

# Build the system
system.build()
```

```
% sst merlin-ember-example.py
Allreduce: ranks 32, loop 1, 1 double(s), latency 5.192 us
Simulation is complete, simulated time: 18.7055 us
```

Hands On with Merlin and Ember

Things to try:

- Change the ember motif being run; possible other motifs to run:
  - Halo3d26
  - sweep3D
- Change some of the network parameters
  - link_bw, input/output latencies, etc
- Use a different routing algorithm
  - minimal, min-a, or valiant
- Try random allocation of the job
- Double the size of the network and add a second job
  - Try different allocations (linear, random, random-linear)

Extending the functionality of Merlin

Creating a new topology for Merlin

Two approaches to adding functionality to merlin

- Add features to the merlin library directly and submit changes for merging
- Create an external element library with the changes and distribute library as a third party plug-in
  - Merlin installs all header files necessary to create a new implementation of its SubComponent APIs

When adding a new topology, there are two main pieces of functionality to implement:

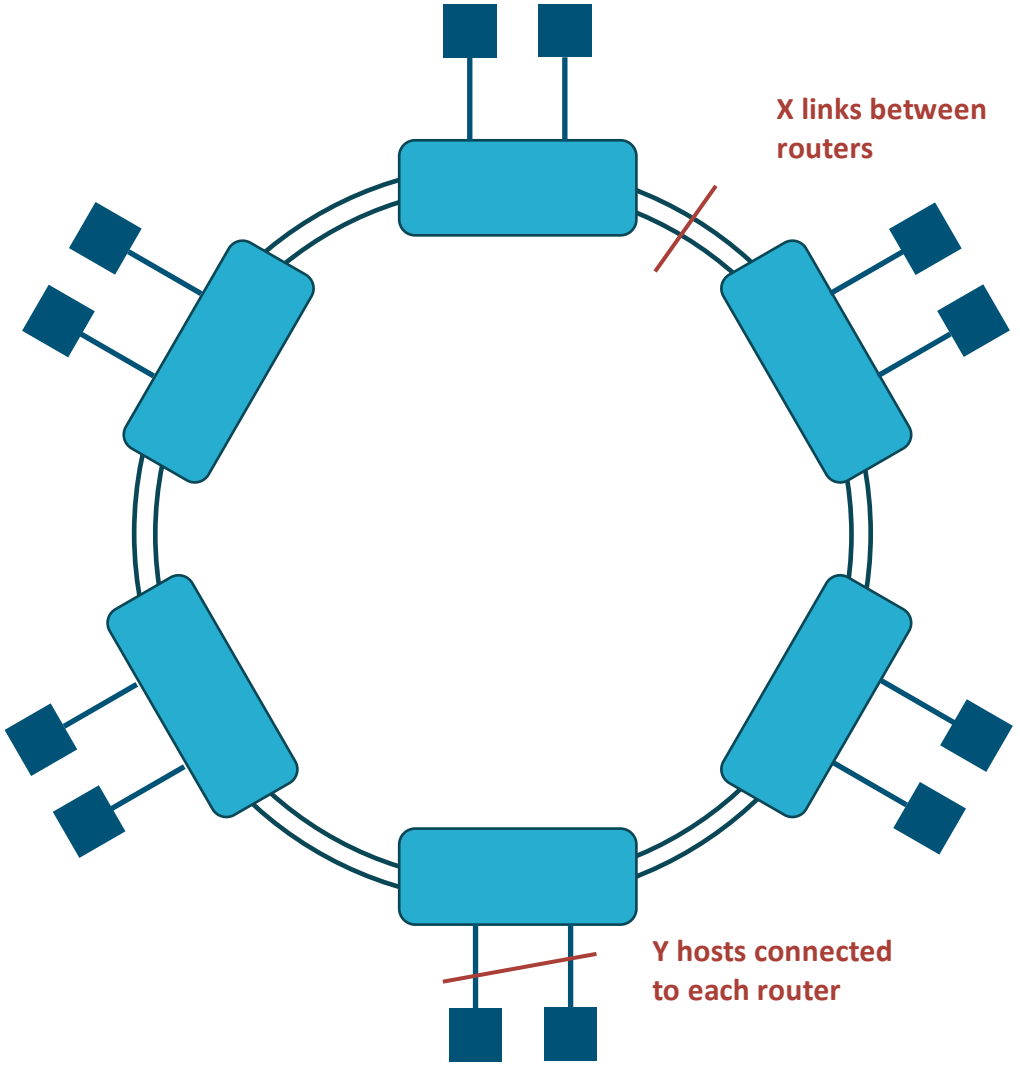
- C++ simulation model inheriting from Merlin::Topology
  - Defined in sst/elements/merlin/router.h
- Python configuration help class inherited from merlin.base.Topology
  - Defined in sst/elements/merlin/pymerlin-base.py
    - Making this Python module available to the interpreter through the ELI requires you to create a C++ class inheriting from SST::SSTElementPythonModule
      - Defined in sst/core/model/element_python.h

Example: Ring Topology

Simple ring topology with multiple links between routers and multiple endpoints connected to each router

Topology Parameters:

- num_routers – total number of routers
- interrouter_links – number of links between routers
- local_ports – number of hosts per router



Let’s look at the code: archimedes element library

Makefile – Makefile to build the libarchimedes.so element library

topo_ring.h – Header file for the ring topology object and events

topo_ring.cc – Source file for the ring topology implementation

- Also includes code to register the archimedes python module with SST core

pyarchimedes.py – Implementation of the archimedes Python module

ring_test.py – Input file to test the ring topology

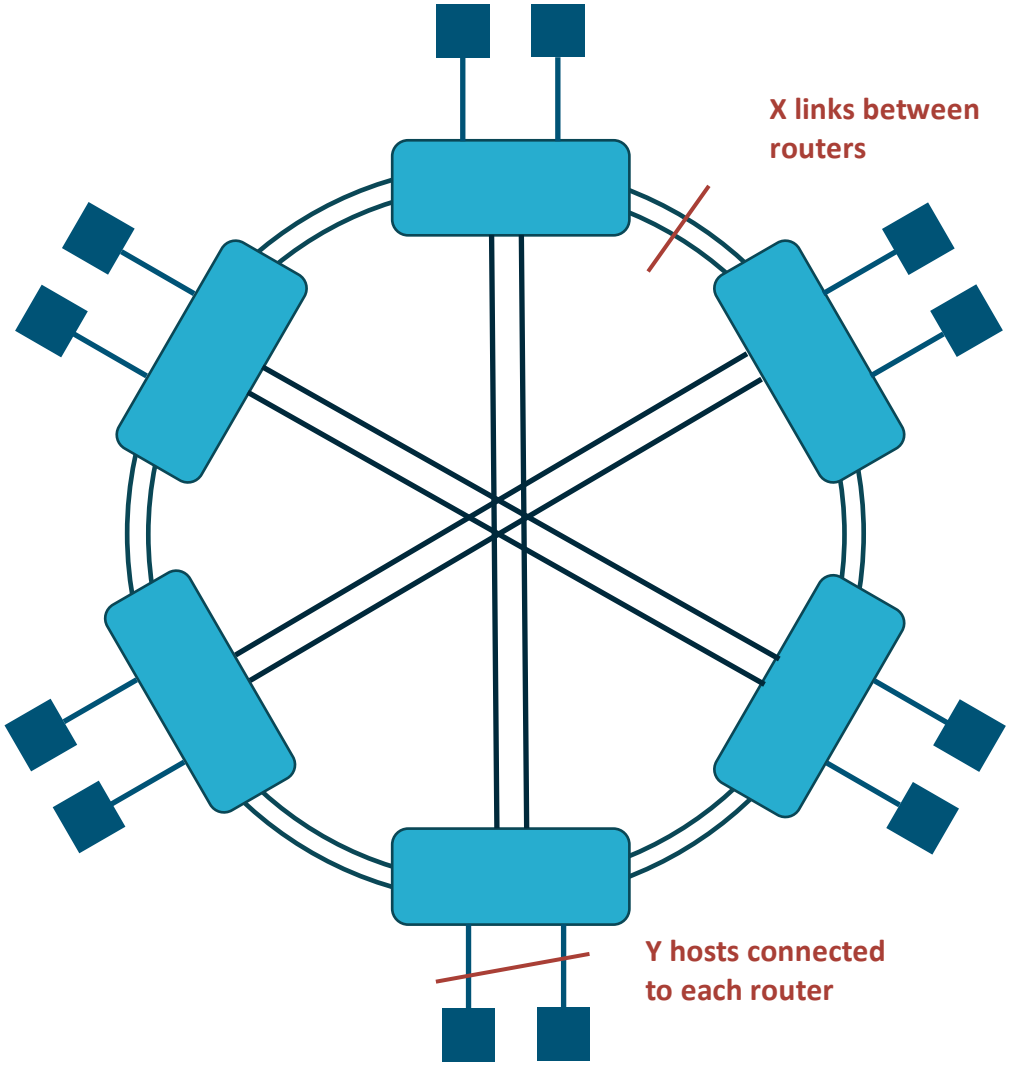
Hands on: Modify topology to include cross links

Try adding cross links to the topology to cut average hop count in half.

- May want to limit topology to even number of routers only

Will need to modify:

- Routing function in SubComponent model
  - Depending on implementation, may need to modify some of the other functions in topo_ring class.
- _build_impl() function in the Python template class



Sandia National Laboratories

Thanks!

Exceptional Service In The National Interest