Sandia National Laboratories

Exceptional service in the national interest

The Structural Simulation Toolkit

The SST SCC Team

Online Presentation

12 August 2025

SAND2025-10051PE

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

U.S. DEPARTMENT OF ENERGY

NNSA

SST SCC Team Members

Patrick Lavin	prlavin@sandia.gov – In-person Lead
Shannon Kinkead	skinkea@sandia.gov – IndySCC Lead
Gwen Voskuilen	grvosku@sandia.gov
Clay Hughes	chughes@sandia.gov
Scott Hemmert	kshemme@sandia.gov

Learning Objectives – Part 1: Introduction to SST

This section of the tutorial will cover the following topics:

1. The basic structure of the SST project
2. How to run a simulation in SST
3. A summary of the components in the SCC problems
4. Performance optimization in SST
5. Where to find more info on SST

References

Websites

- Information on installing SST, and links to everything else you see here
  - http://www.sst-simulator.org/
- Documentation on SST's key interfaces, overview of all the elements, and more!
  - http://sst-simulator.org/sst-docs/
- Code
  - https://github.com/sstsimulator

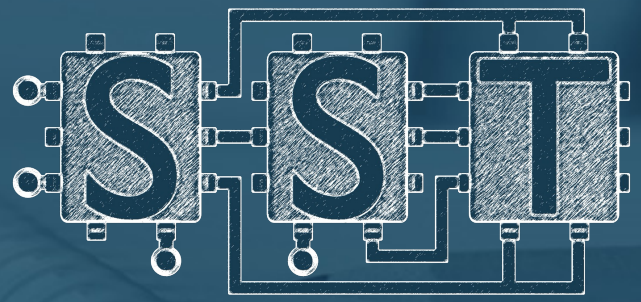
Important Pages

- Configuration File Format: http://sst-simulator.org/SSTPages/SSTUserPythonFileFormat/
- Building SST:
  - Quick : http://sst-simulator.org/SSTPages/SSTBuildAndInstall_15dot0dot0_SeriesQuickStart
  - Detailed: http://sst-simulator.org/SSTPages/SSTBuildAndInstall_15dot0dot0_SeriesDetailedBuildInstructions/

Tutorial

- Our most recent tutorial:
  - https://github.com/sstsimulator/sst-tutorials/tree/master/ipdps2025

SST Overview



Computer Architecture Simulation

Designing computers is expensive

- Apple spent ~$1B on the tape out for the M3 chip
- Lawrence Livermore National Laboratory awarded a $600M contract to HPE for the El Capitan supercomputer (No. 1 Top500)

How do we know a new system is going to perform well? Simulators allow us to:

- Evaluate new architectural ideas (new prefetcher, more memory, new network topology)
- Evaluate how we can change applications to better exploit new hardware features

Why Use SST for simulation?

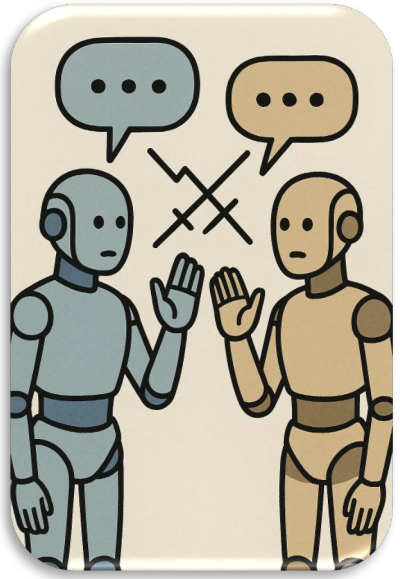
Many computer architecture simulators exist, at many different scales

1. RTL simulators
2. OOO CPU simulators
3. Cache simulators, DRAM simulators
4. Network simulators

However,

- They can’t talk to each other
- They don’t scale

SST was created to solve these problems



The Structural Simulation Toolkit

Goals

- Create a standard architectural *simulation framework* for HPC
- Ability to evaluate future systems on DOE/DOD workloads
- *Use the supercomputers of today to design the supercomputers of tomorrow*

The SST Approach

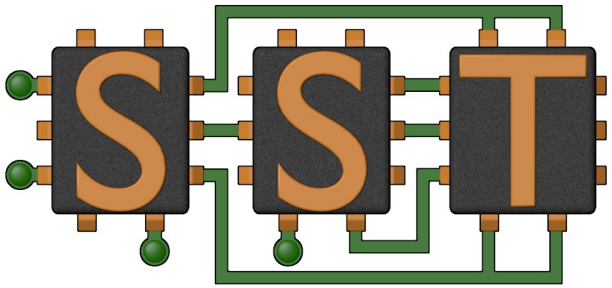
Parallel Discrete-Event Simulator Framework (*SST Core*)

- The backbone of simulation
- Provides utilities and interfaces for simulation components (models)
  - Clocks, event exchange, statistics and parameter management, parallelism support, etc.
- Demonstrated scaling to over 512 processors running a million+ components

Comes with many built-in simulation models (*SST Elements*)

- Libraries of components that perform the actual simulation
- Elements include processors, memory, network, etc.

Written in C++, configured in Python



SST Jargon

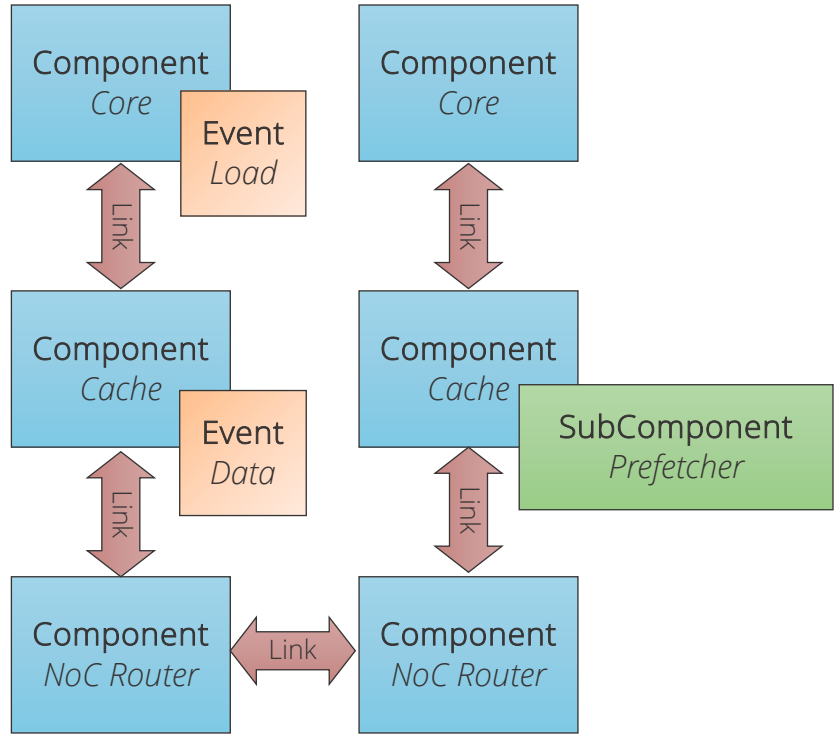
SST simulations are comprised of **components**

Components are connected by **links**

- Every link has a minimum (non-zero) latency
- Components define **ports** which are valid connection points for a link

Components communicate by sending **events** over the links

Components can use **subcomponents** and **modules** for customizable functionality



Element Library

A collection of components, subcomponents, and/or modules

Basic SST Simulation

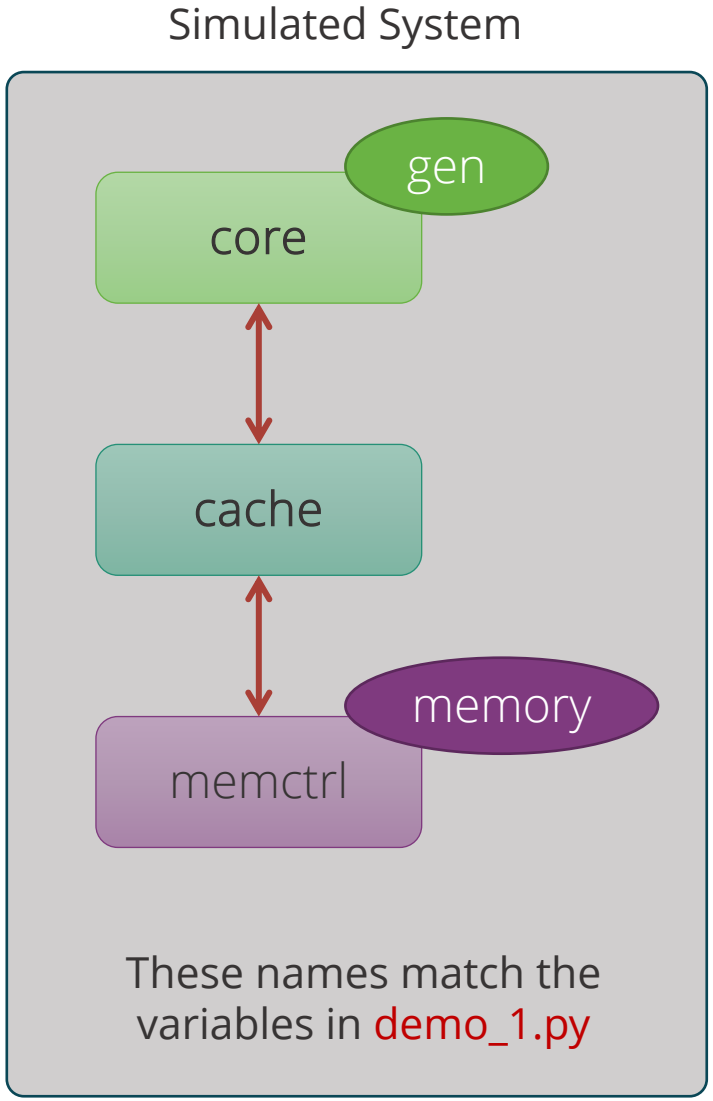
Our First Simulation – demo_1.py

We'll walk through how to configure a simulation and then run it

- Available at: https://github.com/sstsimulator/sst-tutorials/blob/master/ispass2024/exercises/single-node/demo_1.py

Element libraries in our example simulation

- **Miranda** - Simple core model that runs generated instruction streams
  - Generators produce memory access patterns (SubComponent)
- **memHierarchy –** Various cache/memory system related subcomponents and modules
  - Cache (Component) with coherence protocol SubComponent
  - Memory Controller (Component) that loads a memory timing model (SubComponent)



Configuration File: Global SST parameters

Set any global simulation parameters

SST Python API
User-defined string
SST argument

Other options

- Most command line options to SST can be set using setProgramOption()
- We will talk about interesting command line options in the section on performance optimization

```
import sst

##############################
## Define SST core options
##############################
# If this demo gets to 100ms, something
# has gone very wrong!
sst.setProgramOption("stop-at", "100ms")
```

Configuration File: Declare components

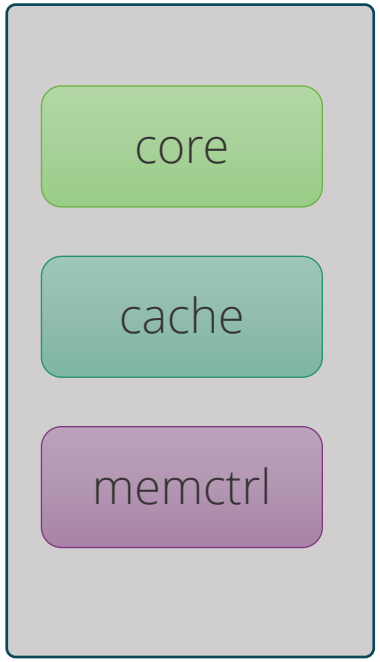
SST Python API
User-defined string
SST argument

Components: `sst.Component("name", "type")`

```
##########################
## Declare components
##########################
core = sst.Component("core", "miranda.BaseCPU")
cache = sst.Component("cache", "memHierarchy.Cache")
memctrl = sst.Component("memory", "memHierarchy.MemController")
```

Component name

Element library

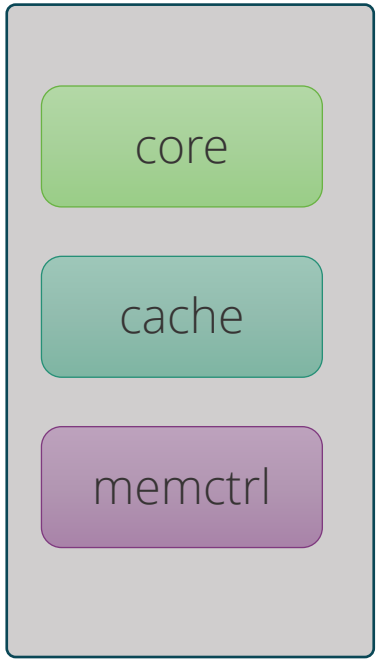


Configuration File: Configure the core

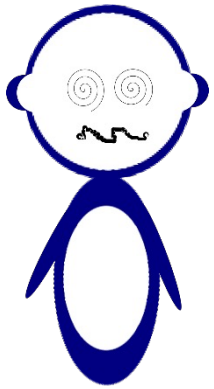
SST Python API
User-defined string
SST argument

Parameters: `addParams({ “parameter” : “value”, … })`

```
##########################################################
## Set component parameters and fill subcomponent slots
##########################################################
core.addParams({
    "clock" : "2.4GHz",
    "max_reqs_cycle" : 2,
})
```



How do I know what the options are?
Or even what elements I can pick from?



Aside: sst-info

Use sst-info to find information about all registered elements.

```
$ sst-info miranda.baseCPU

PROCESSED 1 .so (SST ELEMENT) FILES FOUND IN DIRECTORY(s)
[...]
================================================================================
ELEMENT LIBRARY 0 = miranda ()
Components (1 total)
   Component 0: BaseCPU
      Description: Creates a base Miranda CPU ready to execute an address
generator/access pattern
      ELI version: 0.9.0
      Compiled on: Dec  1 2023 14:33:14, using file: mirandaCPU.h
      Category: PROCESSOR COMPONENT
      Parameters (12 total)
         max_reqs_cycle: Maximum number of requests the CPU can issue per cycle
         (this is for all reads and writes)  [2]
         ...
```

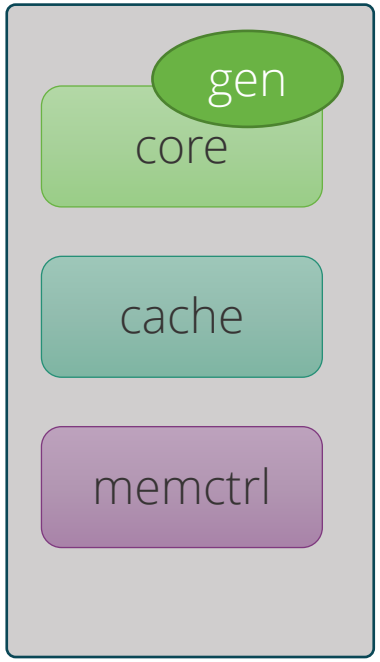
Configuration File: Add a subcomponent

SST Python API
User-defined string
SST argument

SubComponents: setSubComponent("slotname", "type")

- Recall: SubComponent is a *swappable piece of functionality*

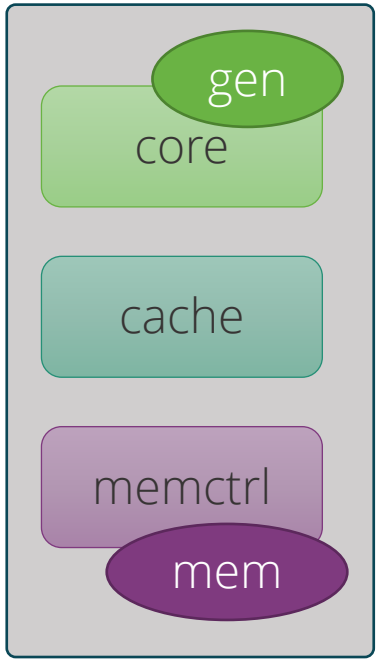
```
gen = core.setSubComponent("generator", "miranda.STREAMBenchGenerator")
gen.addParams({
    "n" : 1000, # Number of array elements
})
```



Configuration File: Configure the cache

SST Python API
User-defined string
SST argument

```
cache.addParams({
    "L1" : 1,
    "cache_frequency" : "2.4GHz",
    "access_latency_cycles" : 2,
    "cache_size" : "2KiB",
    "associativity" : 4,
    "replacement_policy" : "lru",
    "coherence_policy" : "MESI",
    "cache_line_size" : 64,
})
```

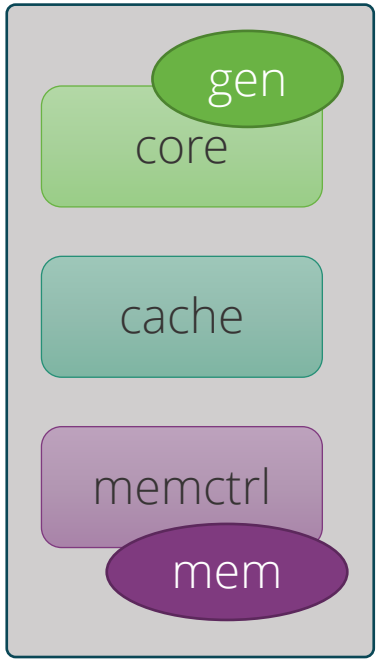


Configuration File: Configure the memory controller

SST Python API
User-defined string
SST argument

```
memctrl.addParams({
    "clock" : "1GHz",
    "backing" : "none", # No real memory values, just addresses
    "addr_range_end" : 1024*1024*1024-1,
})

memory = memctrl.setSubComponent("backend", "memHierarchy.simpleMem")
memory.addParams({
    "mem_size" : "1GiB",
    "access_time" : "50ns",
})
```



Configuration File: Declare and connect

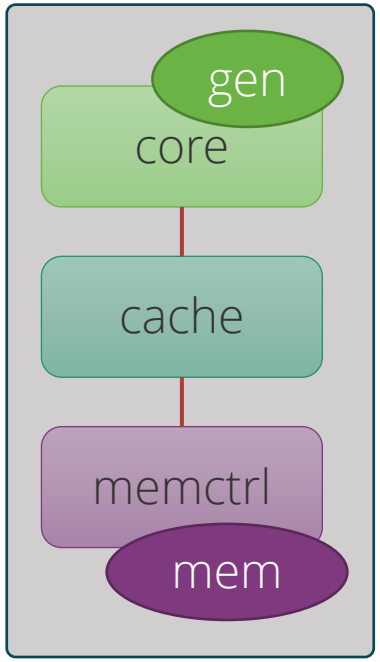
SST Python API
User-defined string
SST argument

Links: `sst.Link("name")`

```
##########################################
## Declare links
##########################################
core_cache = sst.Link("core_to_cache")
cache_mem = sst.Link("cache_to_memory")

##########################################
## Connect components with the links
##########################################
core_cache.connect( (core, "cache_link", "100ps"),
                    (cache, "high_network_0", "100ps") )

cache_mem.connect( (cache, "low_network_0", "100ps"),
                   (memctrl, "direct_link", "100ps") )
```



Running a simulation

Launch simulation

```
$ sst demo_1.py
```

Output

```
Simulation is complete, simulated time:  6.80711 us
```

We probably want more information about what happened though

- Enable statistics!

Running with statistics enabled

Let's enable statistics for all components

- Caches have A LOT of statistics so send the output to a CSV file
- Other options: sst.statoutput*X* where X=
  - console
  - json
  - txt
  - hdf5

```
sst.setStatisticOutput("sst.statoutputcsv")

sst.setStatisticOutputOptions({ "filepath" : "stats.csv" })

sst.setStatisticLoadLevel(5)

sst.enableAllStatisticsForAllComponents()
```

A Tour of SST Elements

SST Element Libraries

Elements are libraries of related components

- Elements must be *registered* with the SST core
  - Tells SST where to find this set of components
  - Includes information on parameters and statistics for each component

SST provides a set of element libraries

- Processor, network, memory, etc.
- Tested for interoperability within and across libraries
- Many are compatible with external “components” such as Ramulator and Spike

You can also register your own elements

- Example: https://github.com/sstsimulator/sst-external-element

Selected Elements

Processors
- **Ariel** – PIN-based
- **Prospero** – trace execution
- **Miranda** – pattern generator
- **Vanadis** – MIPS32 and RV64 pipeline
- GeNSA – Spiking temporal processing unit
- Osseous – RTL simulation

Memory Subsystem
- **MemHierarchy** – caches, directory, memory
- CacheTracer – cache tracing
- Cassini – cache prefetchers
- CramSim – DDR, HBM
- Messier – NVM
- Samba – TLB
- VaultSim – vaulted stacked memory

Network drivers
- **Ember** – communication patterns
- **Firefly** – communication protocols
- **Hermes** – MPI-like driver interface
- Zodiac – trace based driver
- Thornhill – memory models for Ember

Networks/NoCs
- **Merlin** – flexible network modeling
- **Kingsley** – mesh NoC
- Shogun – crossbar NoC

Accelerators
- Balar – GPGPU-Sim interface
- Llyr – spatial compute

Others
- sst-macro → Mercury
- simpleElementExample
- simpleSimulation

Vanadis: OOO Processor

$ sst-info vanadis.VanadisNodeOS

- MPIS32 and RV64 compatible processor model
- OOO model
  - Configure micro-architectural details
    - Instruction fetch/decode/retire rate
    - ROB size
  - Branch prediction
- Multi-threaded cores
- Musl libc used to cross-compile programs
  - Also tested with clang
- System-call emulation

MemHierarchy: Memory system

```
$ sst-info memHierarchy | grep " Component"
```

Collection of interoperable memory system elements

- Caches
- Directories
- Memory controllers
  - Interfaces to memory models (DDR, HBM, HMC, NVM, etc.)
- Scratchpads
- NoC (network-on-chip) interfaces
- Buses

Components are cycle-accurate/cycle-level

Capable of modeling modern cache and memory subsystems

Merlin: Network simulator

$ sst-info merlin

Low-level 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,

Many ways to drive a network

- Simple traffic generation models
  - Nearest neighbor, uniform, uniform w/ hotspot, normal, binomial
- *MemHierarchy*
- Lightweight network endpoint models (*Ember* – coming up next)
- Or, make your own

Ember: Network Traffic Generator

$ sst-info ember

Light-weight endpoint for modeling network traffic

- Enables large-scale simulation of networks where detailed modeling of endpoints would be expensive

Packages patterns as *motifs*

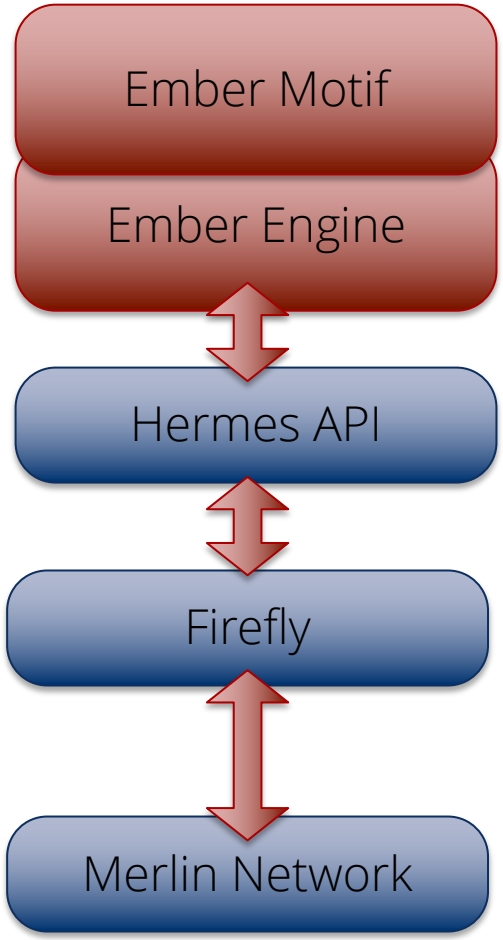
- Can encode a high level of complexity in the patterns
- Generic method for users to extend SST with additional communication patterns

Intended to be a driver for the Hermes, Firefly, and Merlin communication modeling stack

- Uses Hermes message API to create communications
- Abstracted from low-level, allowing modular reuse of additional hardware models

Ember: Overview

$ sst-info ember



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.

Performance
Optimization

SST Lifecycle
Monitoring
Optimizing

Parallelism

SST was designed from the ground up to enable scalable, parallel simulations

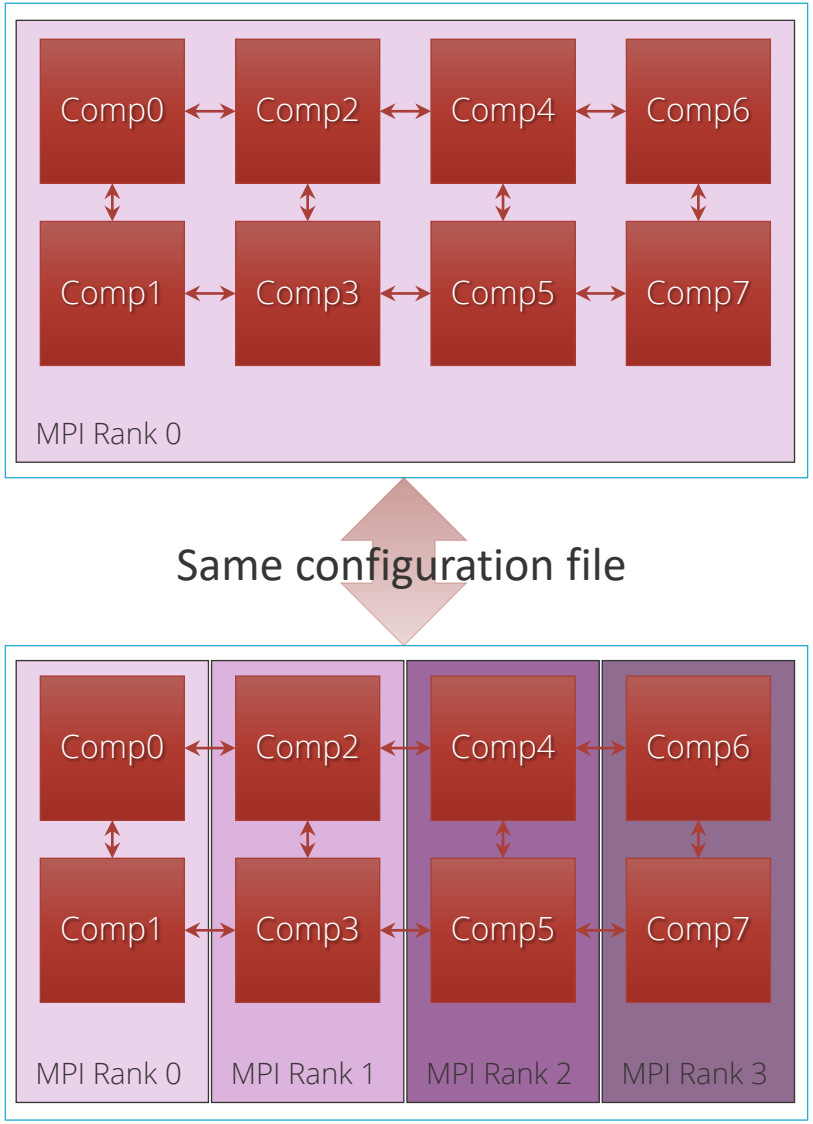
Components distributed among MPI ranks/threads

- Minimum link latency of *cut links* controls synchronization rate

```
# Two ranks
$ mpirun -np 2 sst demo_1.py

# Two threads
$ sst -n 2 demo_1.py

# Two ranks with two threads each
# This will give a warning since we only
# have 3 components across 4 ranks/threads
$ mpirun -np 2 sst -n 2 demo_1.py
```



Partitioning

```
mpirun –np <ranks> sst –n <threads_per_rank> sst_input.py
```

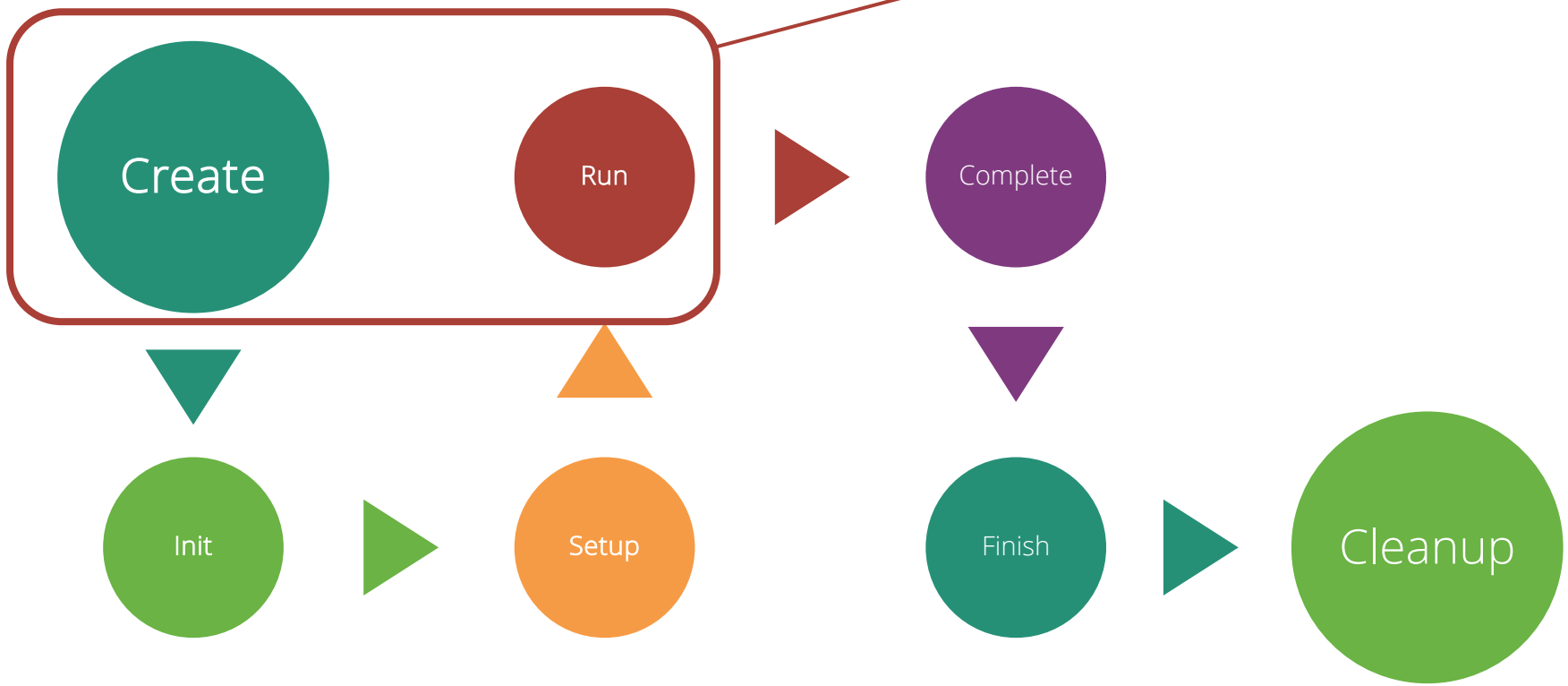
Use `sst-info sst` to see built-in partitioners

- linear (default), roundrobin, self, simple
- *self* requires specifying the partition in the input file
- Controlling partition from the input file
  - `link.setNoCut()` prevents link from being cut
  - `component.setRank()` puts a component on a specific rank/thread *(self* partition only)
  - `component.setWeight()` adds a relative weight, used by some partitioners

Generally…

- Load-balance compute intensive components across threads/ranks
- Maximize work done on-partition between synchronizations

The SST Lifecycle



Typically consume the most time and memory

Create:

- Generate, partition, and distribute the simulation graph, construct simulation
- Size of simulation (# components) matters

Run:

- Simulation execution
- Partition matters for synchronization frequency

All stages are parallel although “Create” is only partially parallel by default.

https://sst-simulator.org/sst-docs/docs/guides/concepts/lifecycle

Command line options for performance and partitioning

- `--print-timing-info` – Prints wall-clock time and peak memory during lifecycle stages
- `--heartbeat-wall-period, --heartbeat-sim-period` – Prints some performance info at a regular wall or simulation time frequency but *only during run stage*
- `--run-mode=init` – Runs SST up to the run phase and exits
- `--output-partition` – Write out the generated partition
- `--partitioner=PARTITIONER` – Select the partitioner SST should use.
- `--parallel-load` – Execute the create stage entirely in parallel using a pre-partitioned parallel input file (can be generated by --parallel-output).

References

Websites

- Information on installing SST, and links to everything else you see here
  - http://www.sst-simulator.org/
- Documentation on SST's key interfaces, overview of all the elements, and more!
  - http://sst-simulator.org/sst-docs/
- Code
  - https://github.com/sstsimulator

Important Pages

- Configuration File Format: http://sst-simulator.org/SSTPages/SSTUserPythonFileFormat/
- Building SST:
  - Quick : http://sst-simulator.org/SSTPages/SSTBuildAndInstall_15dot0dot0_SeriesQuickStart
  - Detailed: http://sst-simulator.org/SSTPages/SSTBuildAndInstall_15dot0dot0_SeriesDetailedBuildInstructions/

Tutorial

- Our most recent tutorial:
  - https://github.com/sstsimulator/sst-tutorials/tree/master/ipdps2025