SST Practice Problems

Date: 4 September 2025

Introduction

The Structural Simulation Toolkit (SST) is a conservative parallel discrete event simulator (PDES) designed for simulating computer systems. It is composed of two main projects, SST-Core, which implements core PDES infrastructure such as clocks, synchronization, events and links, and SST-Elements, which is a collection of models of computer components, such as cores, caches, and routers, that run on top of SST-Core.

This document contains basic information about the performance characteristics of SST as well as a few example problems to help you prepare for the 2025 SC Student Cluster Competition.

Characteristics of the Code

In SST, the simulation is made up of *components* that communicate with each other by sending *events* across *links*. Components may run in different threads or processes, but they must occasionally synchronize with each other to ensure they do not get too far ahead of connected components. For example, if components A and B are connected by a link with a minimum latency of 1 μs and are located on different ranks, then those ranks must synchronize every 1 μs of simulated time to make sure that one component isn't more than 1 μs behind the other. If component A were to fall this far behind, it may generate a message meant to be delivered in the past of component B. This situation is not allowed in conservative PDES algorithms.

Synchronization is a fundamental bottleneck of PDES. If components in a simulation are connected by low-latency links, they will need to synchronize more frequently. If they do not do much work between synchronization points, it will be hard to scale the code. For this reason, SST simulations of detailed core models, which include caches with low latency links and little work required per lookup, will parallelize more poorly than network simulations, which include much longer latency links relative to the amount of work done.

Internally, SST represents all the components and links as a graph. When running on multiple threads or processes, SST must perform graph partitioning to distribute the components among threads and ranks. This process can be time consuming and can greatly impact performance. Ideally, components connected by shorter latency links are placed on the same rank to reduce the frequency of synchronization, but as we know, graph partitioning is NP-hard, so it is not feasible to get an optimal solution in most cases.

Installing SST

Follow the instructions on the SST website to install SST-Core and SST-Elements.

In the course of creating these problems, we identified a bug that needed to be fixed. Because of this, you'll need to pull the repos from the official `sstsimulator` GitHub page. You should install the `devel` branches of both repos. Specifically, the following commits are what we have used for testing.

- SST-Core: Repo, Commit: `546dda2`
- SST-Elements: Repo, Commit: `22504be`

Use the instructions on the SST website to build and install SST. Link.

Practice Problems

1. Tutorial Content

The SST team has run a number of tutorials. The tutorial we presented to the SCC teams was created from one of them. The slides and example inputs are located in the `sst-tutorials` GitHub repo. You should ignore the mention of containers in the tutorial, as you should have installed SST on your machine.

- Tutorial slides: Link
- Inputs: Link

Goal: Run the examples found in the first 41 slides. Understand the output of SST.

2. Running Vanadis

We are having some issues with the Vanadis toolchain, specifically regarding OpenMP programs. While these instructions should work for the included single-threaded inputs, we will provide more guidance on this practice problem soon.

Vanadis is an element library that contains components used to simulate a pipelined, out-of-order CPU core. We have provided you with `arch/example.py`, which is an SST input file that describes a multicore processor. Vanadis supports the MIPS and RISC-V instruction set architectures, but you are likely using an x86-64 or Arm machine for the competition. This means you will need to cross-compile any applications that you intend to run in Vanadis. To build the cross compiler, follow the instructions on this page in the section "LLVM + RISCV GNU Toolchain" `https://sst-simulator.org/sst-docs/docs/elements/balar/QuickStart#llvm--riscv-gnu-toolchain`.

Once you have the cross-compiler, use the included Makefile to build the programs in `src/`. They can then be used as inputs to the simulator.

Run the simulation as follows:

```
sst example.py -- <executable>
```

Goals: Build the LLVM+RISCV toolchain. Compile the included source files. Run `example.py` on each. Examine the affects of changing the parameters in `example_params.py`. *Note that a recently discoverd bug affects the instructions per cycle of Vanadis. Focus on parameters that are not for the core. Do not edit parameters below the line marked STOP.*

3. Running Ember and Merlin

We have included a network simulation input named `network/scc-network.py`. This input models MPI jobs running on a dragonfly interconnect. This file has been included for you to explore the scalability of SST. A number of knobs are available, which you can view with

```
sst scc-network.py -- -h
```

You should also reference the tutorial the SST team gave, which covered some `sst` options that affect performance.
Goal: Explore how different input parameters to the script affect the scalability of SST.