REV: SCALABLE HPC WORKLOAD SIMULATION USING RISC-V IN SST

John Leidel

Chief Scientist, Tactical Computing labs

OUTLINE

- RISC-V for HPC
- What is SST?
- What is Rev?
- Rev Architecture
- Rev Scalability

Tactical Computing Labs

RISC-V FOR HPC

RISC-V for HPC

- US DoE, DoD, NASA, EU and others are exploring the use of RISC-V for primary and accelerator compute mechanisms for HPC and AI
- No current hardware platforms to experiment with design tradeoffs!
- No current platforms to do software performance experiments!
- ***What about simulation??***

RISC-V Simulation Landscape

- Spike (RISC-V golden model)
- Qemu
- Gem5

- Little/no support for full system simulation!

+

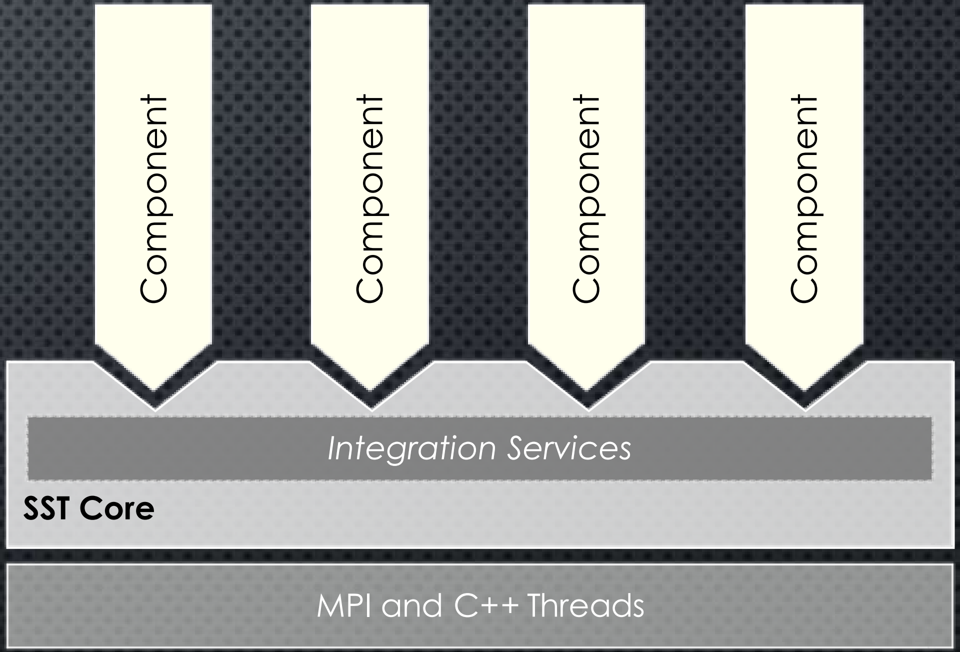
Use our current HPC to simulate the next HPC!

Tactical Computing Labs

WHAT IS SST?

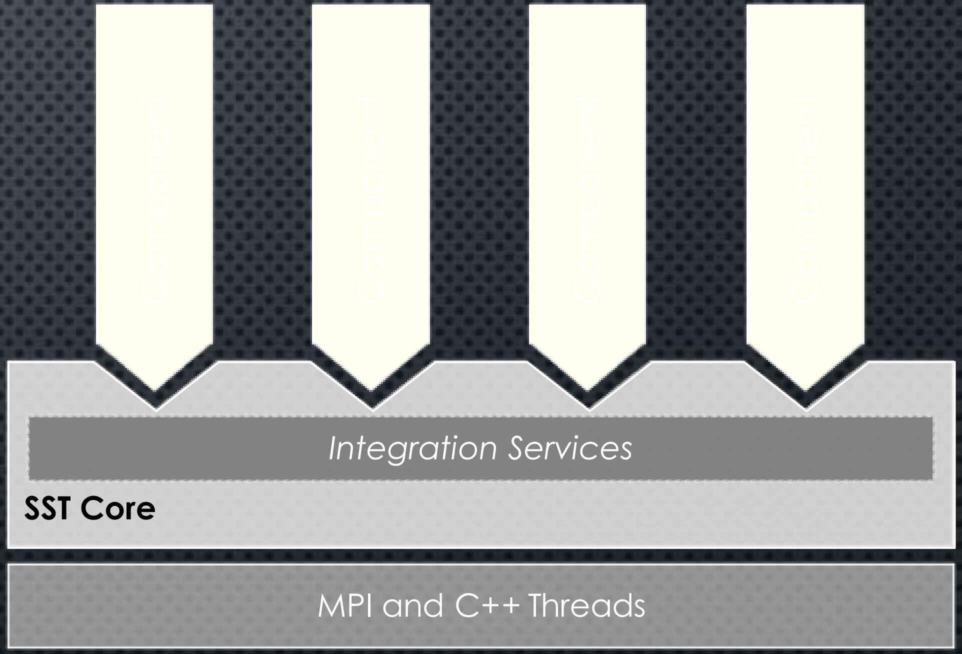
STRUCTURAL SIMULATION TOOLKIT (SST)

- SST is a PDES Framework
- Divided into two libraries
- SST Core
  - Tracks simulation time
  - Transports messages between components
  - Simulation creation and teardown
  - Provides services:
    - Statistic tracking
    - Debug
    - Threading (some)
    - Memory Management (some)
  - Presents an API
  - Likely unnecessary for you to modify
- SST Elements
  - Library of components
  - What you will *actually* use to sim

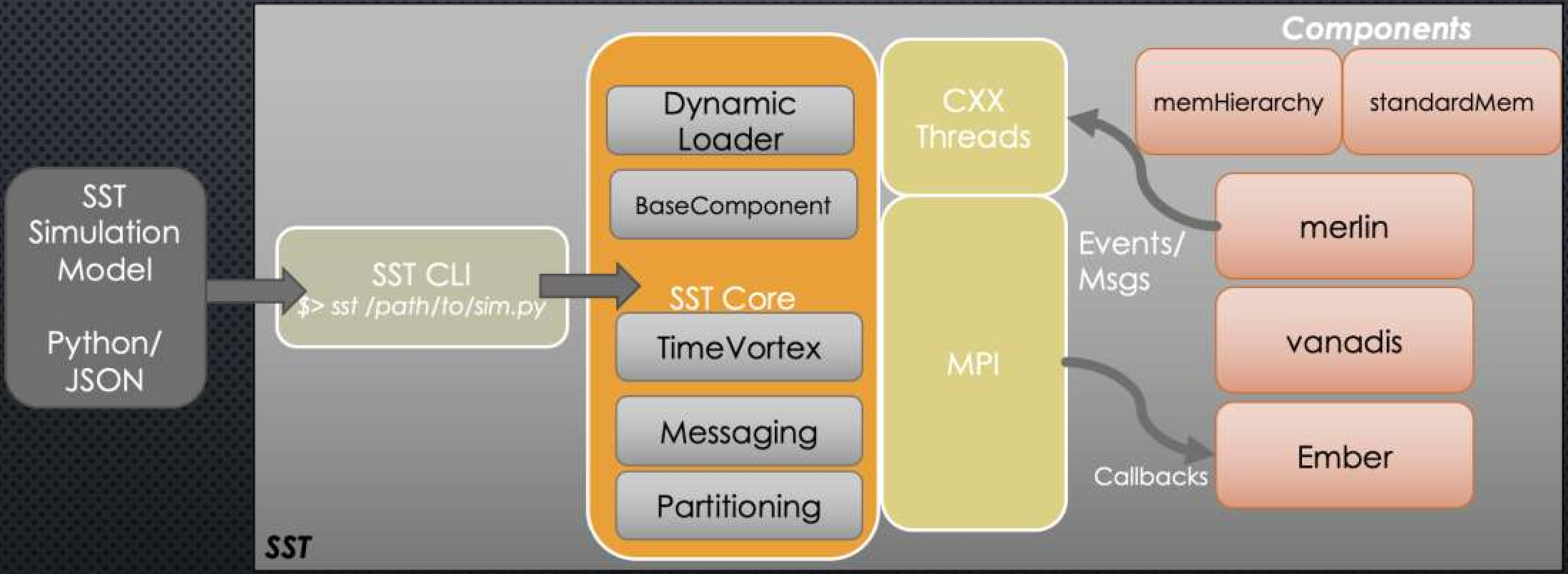


Structural Simulation Toolkit (SST) (cont)

- SST Elements
  - Library of components
  - What you will ***actually*** use to construct your simulation
- Large library of fundamental components
  - CPU models
    - Trace and execution driven
  - Memory models
    - Cache, DRAM, HBM, HMC
  - Interconnect models
    - NoC
    - HPC networks
- Component APIs well defined to allow composition of larger systems
- Create full custom elements for any simulation



SST INTERNAL ARCHITECTURE



Tactical Computing Labs

WHAT IS REV?

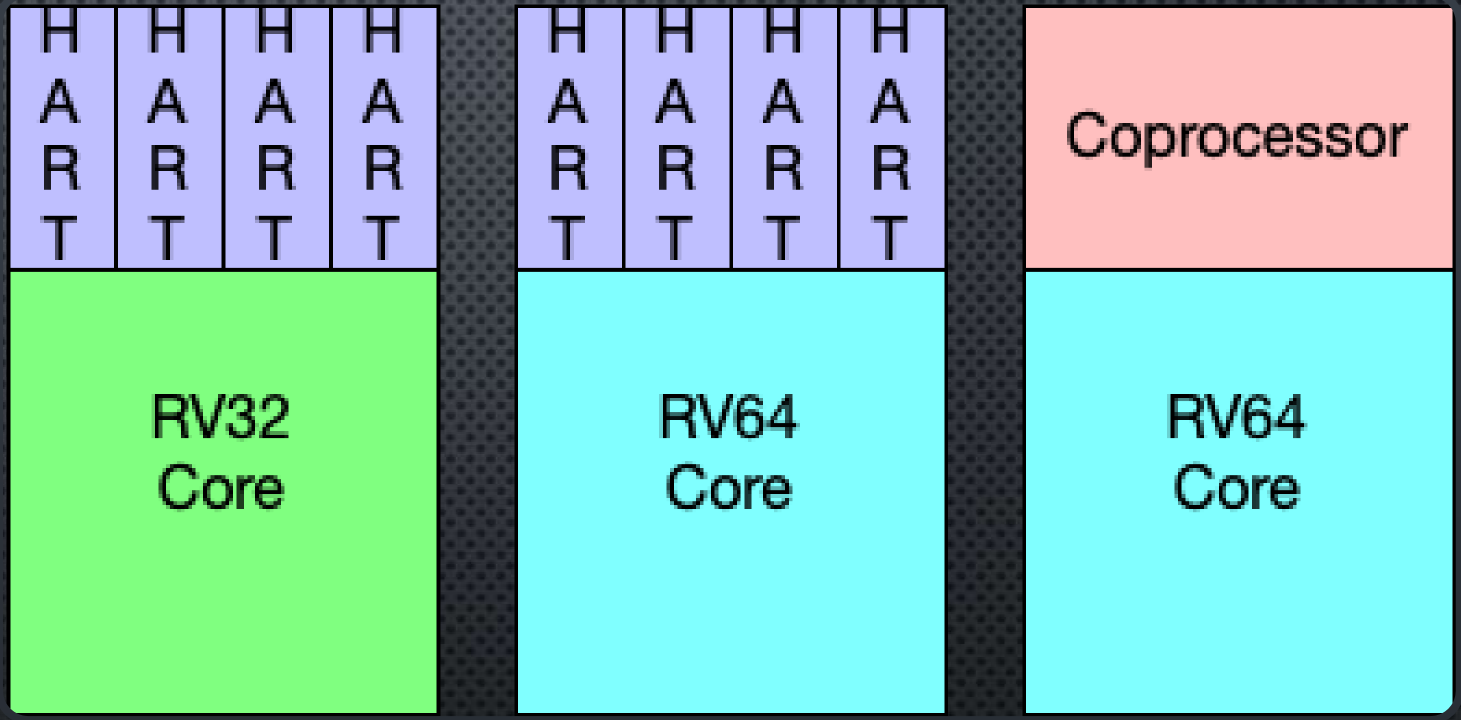
What is Rev?

- Rev is a cycle-base CPU simulation component that directly interfaces with Sandia's SST PDES
  - Stand-alone shared library
- Bare metal simulation infrastructure
- Loads & executes ELF binaries (not traces!)
- Supports interfacing with external memory components (DDR, HBM, etc)
- Supports interfacing with external network components (Merlin)
- Reasonable instruction performance
  - ~3MIPS

https://github.com/tactcomplabs/rev

REV FEATURES

- Support for RV32 and RV64G XLENs
- Supports extensions:
  - Open source: IMAFDC, Zicbom
  - Closed source: vector
- Support for multi-core, multi-hart and multi-core + multi-hart
- Support for heterogeneous SoC's
- Support for user-defined coprocessors



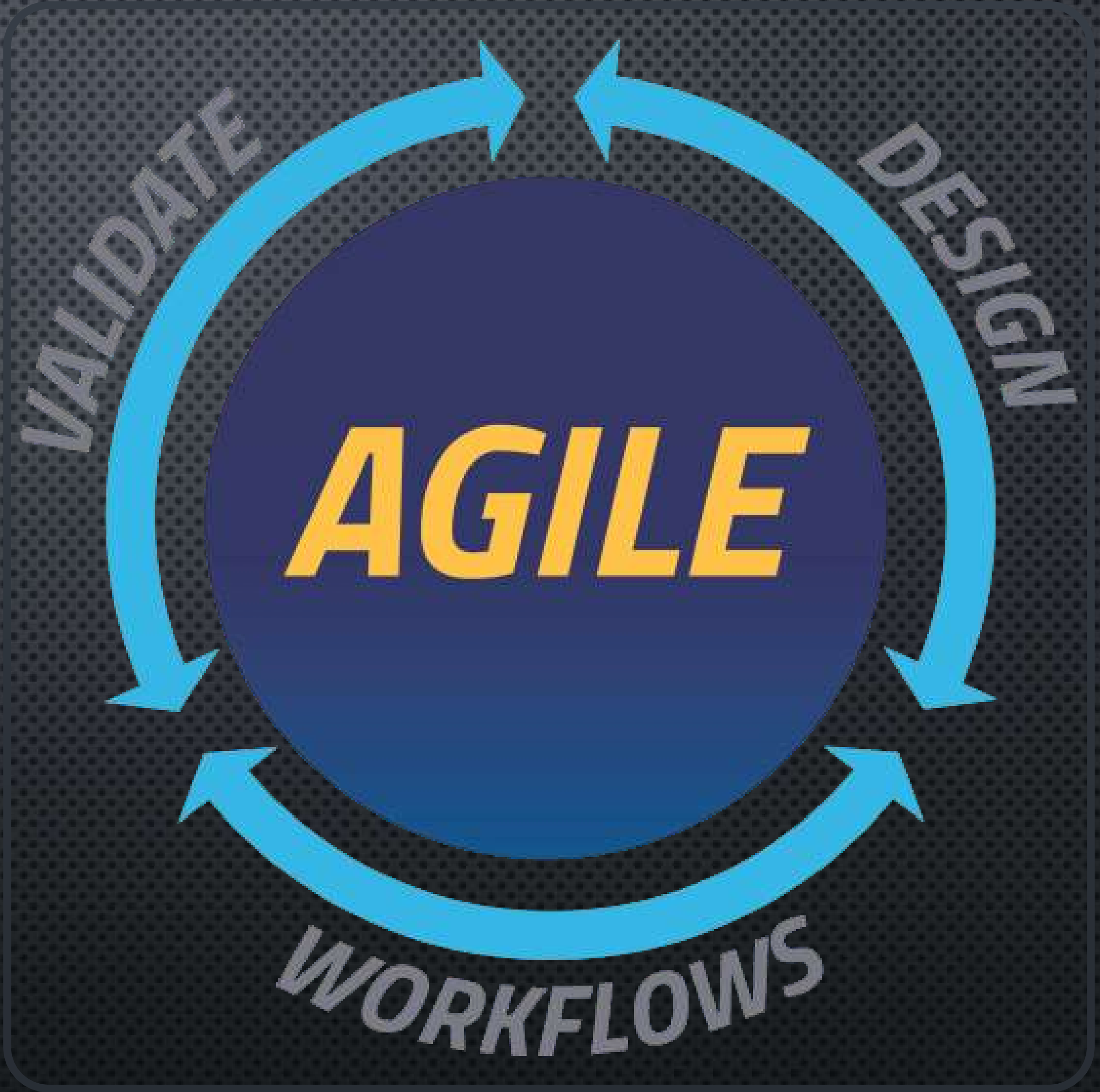
REV EXTENSIBILITY

- Extensions are defined using a well-defined table-generated interface
  - Tables feed the crack+decode interface as well as the individual instruction implementations
- Adding extensions is a well-defined, documented process
- Extensions are not required to handle explicit decoding, hazarding or other arbitration logic
- Each instruction can reside in the pipeline for a user-defined period of time (can be set dynamically at runtime)

- User-defined coprocessors can be loaded as shared library objects in order to implement non-standard RISC-V functionality
- Coprocessors are handled via unimplemented instruction traps
- Coprocessors can support non-standard RISC-V encoding, but maintain access to register state and memories

REV EXTENSIBILITY

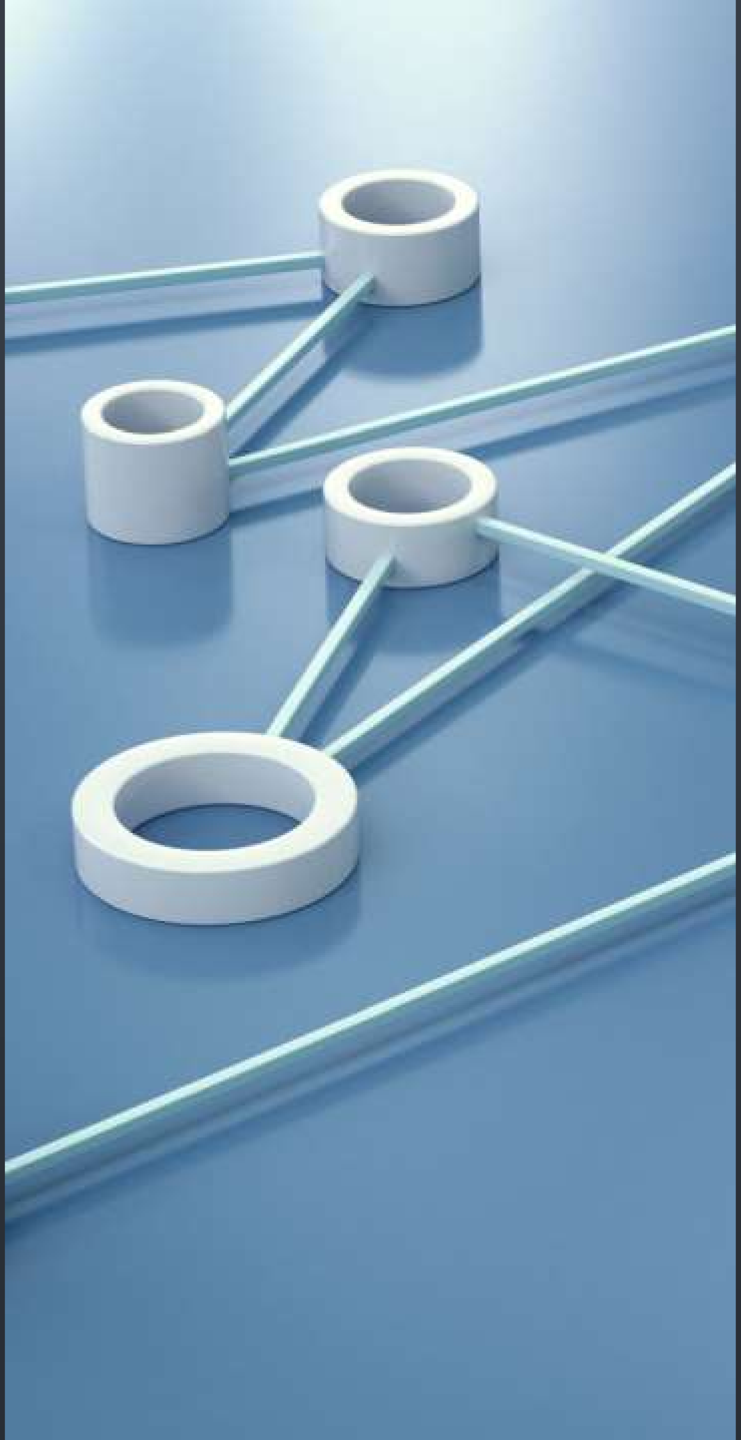
- Rev has been utilized for the IARPA AGILE program to simulate large-scale HPC systems
- We have constructed models that interface with custom coprocessors, external accelerators, NoCs, RDMA networks and complex memory hierarchies
- Initial scalability tests have evolved to simulations exceeding 78,000 Rev CPUs connected to memories and networks
    - Largest SST simulations to date
    - Largest RISC-V simulations to date



https://www.iarpa.gov/research-programs/agile

Rev Extensibility

- Rev has also been utilized to study how various different styles of hardware faults may affect performance, reliability and numerical correctness
- Rev supports injecting random faults into the pipeline logic, register file(s) and/or the memory subsystem with single or multi-bit errors
- This work was done in conjunction with NASA in order to study how well rad-hard or semi-rad-hard RISC-V devices would perform in spaceflight operations



REV & NETWORKING

- We have integrated Rev with the existing SST Merlin network simulation components for a variety of projects
  - Individual projects generally call for specific features/requirements
- Rev contains a basic RDMA NIC infrastructure (RevNIC) that permits users to communicate over arbitrary network topologies
  - The actual definition of network packet contents is currently implementation-specific
- We have experience integrating a variety of compiled runtime libraries using this path
  - MPI, OpenSHMEM, Actors runtime, etc

Tactical Computing Labs

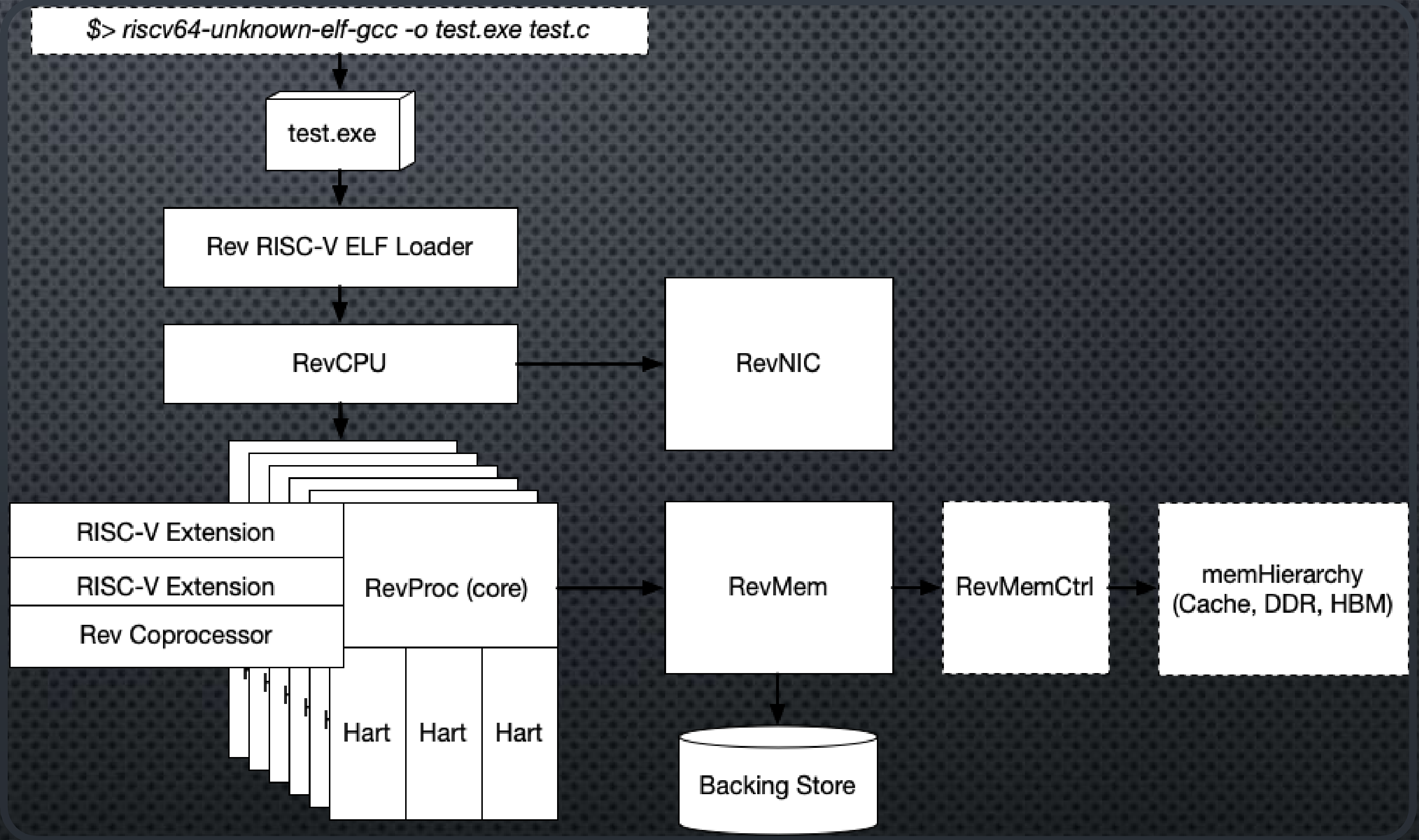
REV ARCHITECTURE

REV ARCHITECTURE OVERVIEW

- Rev is written in C++-17 using the standard set of SST component/subcomponent interfaces
- Rev is composed of multiple internal classes to support:
  - Binary loading (ELF)
  - Instruction (extension) implementations
  - Pipelined execution
  - Backing memory storage
  - Hazarding
  - System calls & threading

Rev Architecture

- Rev executes within the SST simulation framework
- Accepts ELF binary payloads compiled with GCC+Newlib or LLVM+MUSL
- Multiple cores (procs) can be configured, each with unique RISC-V extensions
- Each core can have multiple harts
- The default memory behavior is a simple (fast) delay timing model (RevMem)
- We also support cycle-based cache and memory timing using memHierarchy
- We support an open source, basic NIC device for communicating with Merlin networks



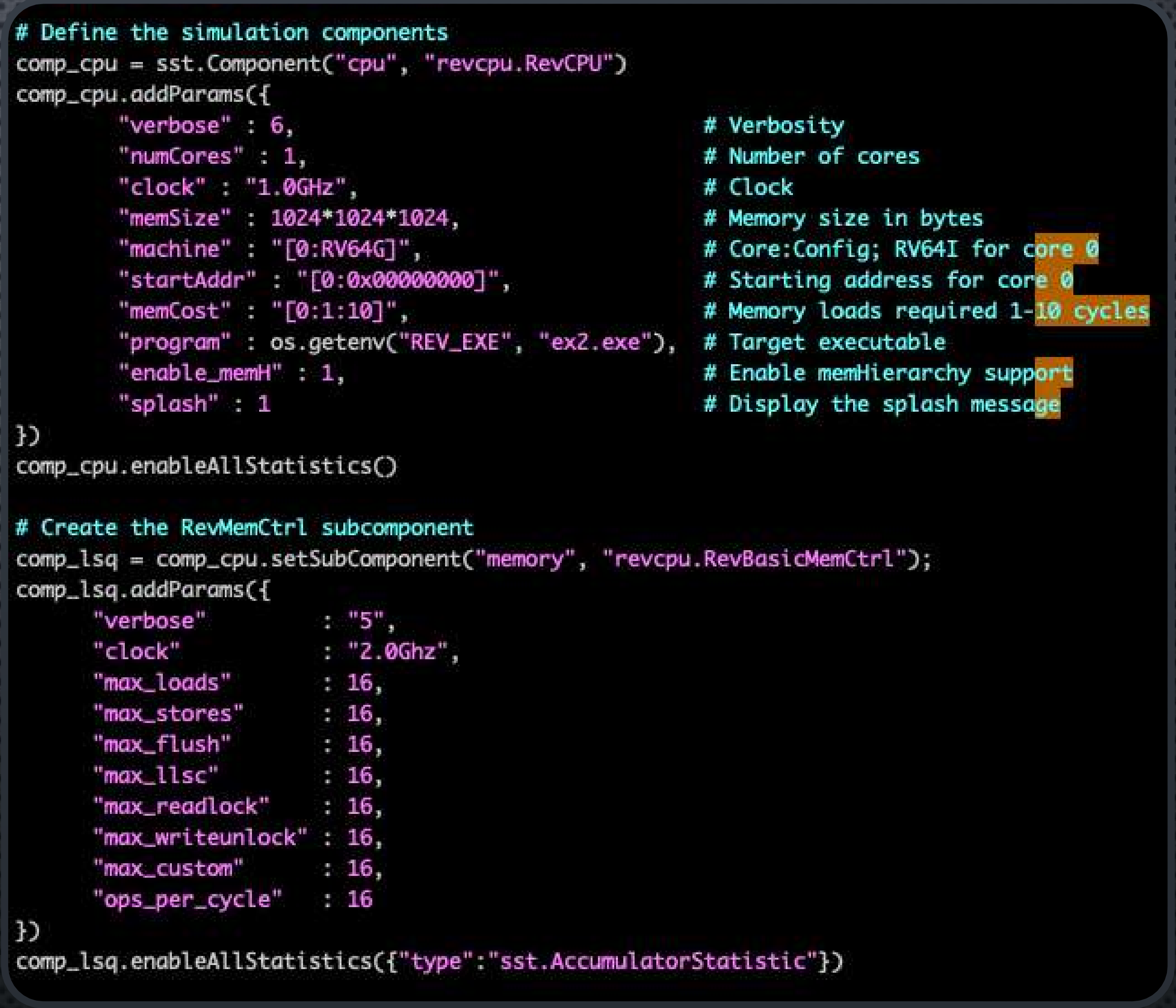
```
$> riscv64-unknown-elf-gcc -o test.exe test.c
```

Rev Architecture Cont.

- Rev has been tested using a variety of C/CXX applications, benchmarks, etc
    - We do not support executing native Python or other interpreted languages
- Each instance of Rev is single threaded
    - Multiple instances of rev can be executed in parallel across CXX threads and/or MPI
- System calls are supported
    - Standard system calls marshal data to the host system for I/O, device access
    - Some system calls require shim interfaces in the user code
    - Additional system calls can added to support RDMA, MMIO, etc
- Rev contains a coprocessor interfaces to build support for non-standard extensions, instructions, etc
    - Users can define their own coprocessors w/o directly modifying the core Rev source base

Configuring REV

- REV is instantiated using the same SST Python/JSON configuration
- Users can set the number of cores per CPU, Harts per core, the core configuration and the executable parameters
  - Program execution can start from arbitrary addresses, symbols, *_start* or *main*
- Users can also inject per-instruction timing information (retire latency), caching hierarchies and define external memories

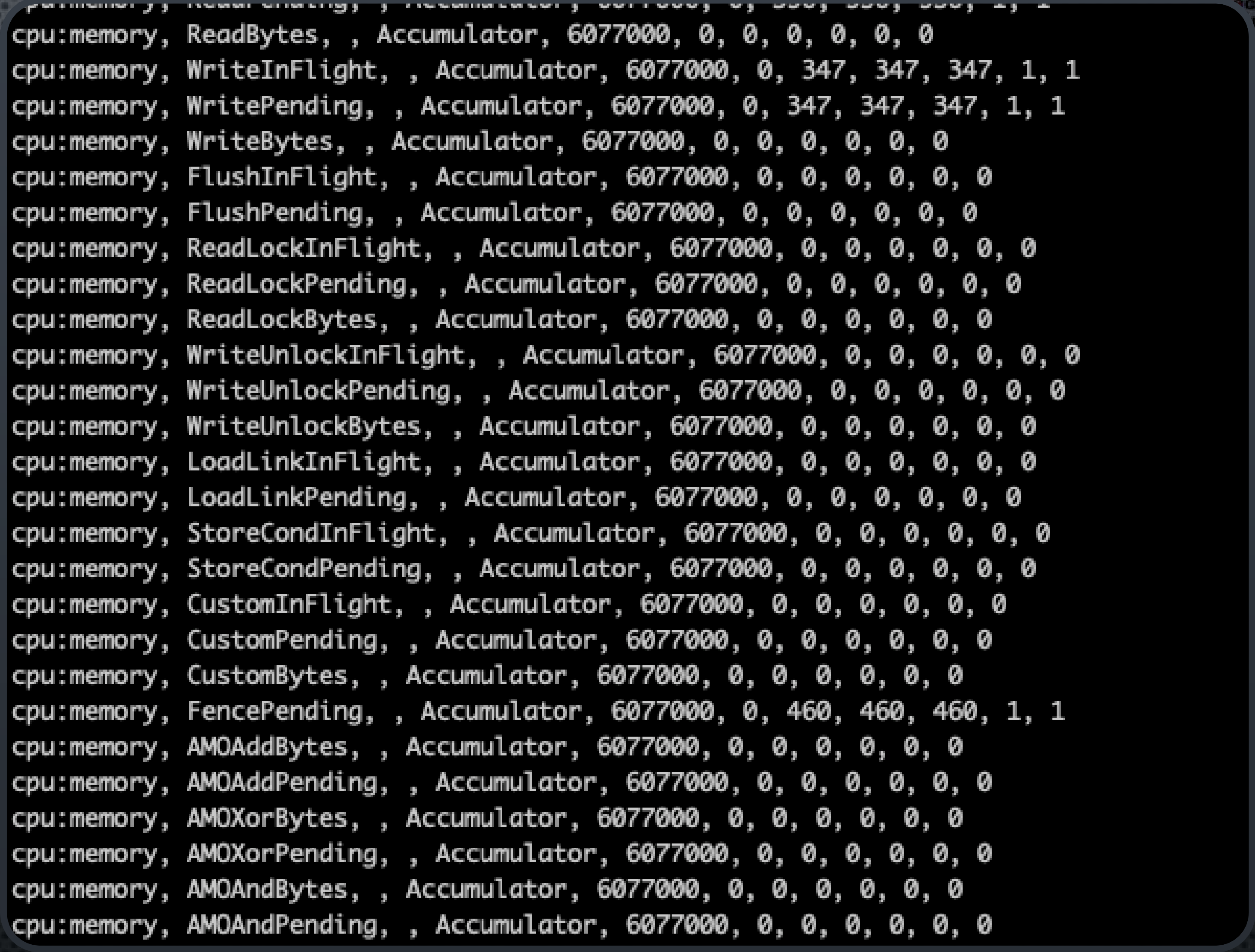


```
# Define the simulation components
comp_cpu = sst.Component("cpu", "revcpu.RevCPU")
comp_cpu.addParams({
        "verbose" : 6,                                        # Verbosity
        "numCores" : 1,                                       # Number of cores
        "clock" : "1.0GHz",                                   # Clock
        "memSize" : 1024*1024*1024,                           # Memory size in bytes
        "machine" : "[0:RV64G]",                              # Core:Config; RV64I for core 0
        "startAddr" : "[0:0x00000000]",                       # Starting address for core 0
        "memCost" : "[0:1:10]",                               # Memory loads required 1-10 cycles
        "program" : os.getenv("REV_EXE", "ex2.exe"),          # Target executable
        "enable_memH" : 1,                                    # Enable memHierarchy support
        "splash" : 1                                          # Display the splash message
})
comp_cpu.enableAllStatistics()

# Create the RevMemCtrl subcomponent
comp_lsq = comp_cpu.setSubComponent("memory", "revcpu.RevBasicMemCtrl");
comp_lsq.addParams({
      "verbose"         : "5",
      "clock"           : "2.0Ghz",
      "max_loads"       : 16,
      "max_stores"      : 16,
      "max_flush"       : 16,
      "max_llsc"        : 16,
      "max_readlock"    : 16,
      "max_writeunlock" : 16,
      "max_custom"      : 16,
      "ops_per_cycle"   : 16
})
comp_lsq.enableAllStatistics({"type":"sst.AccumulatorStatistic"})
```

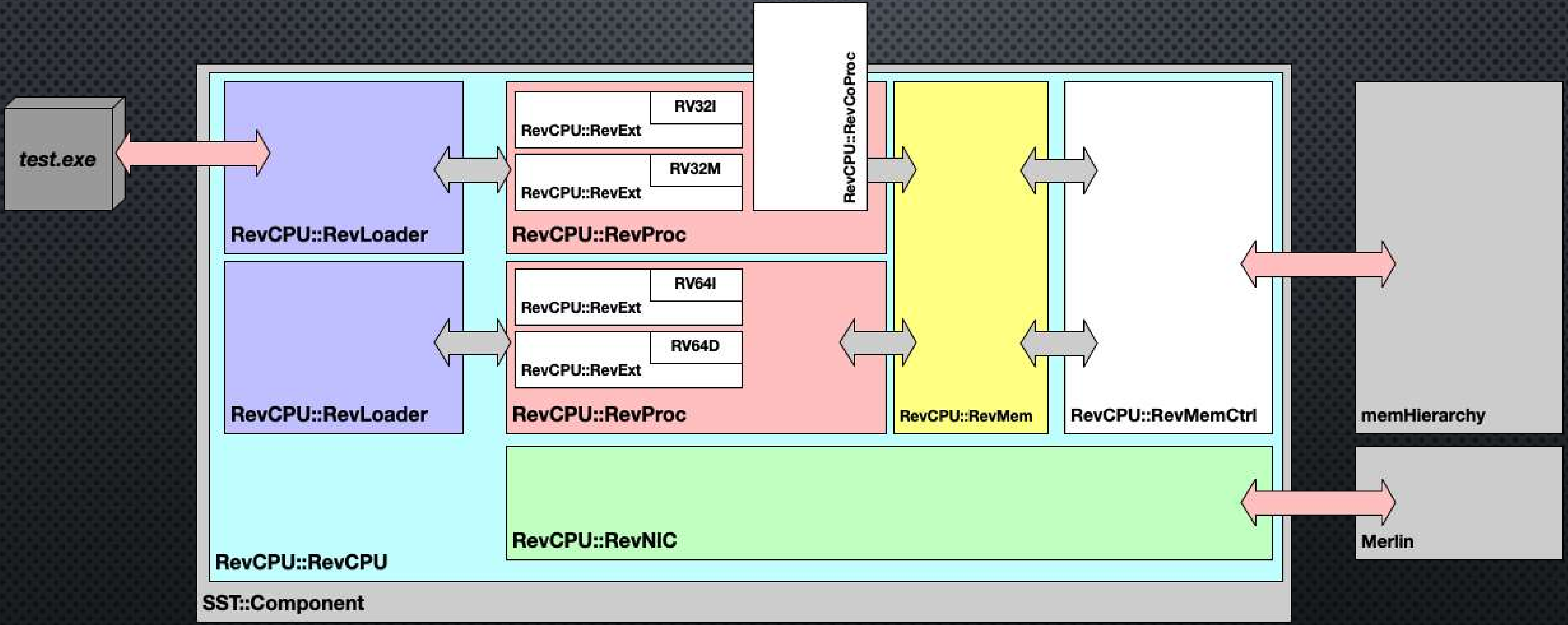
Gathering Statistics

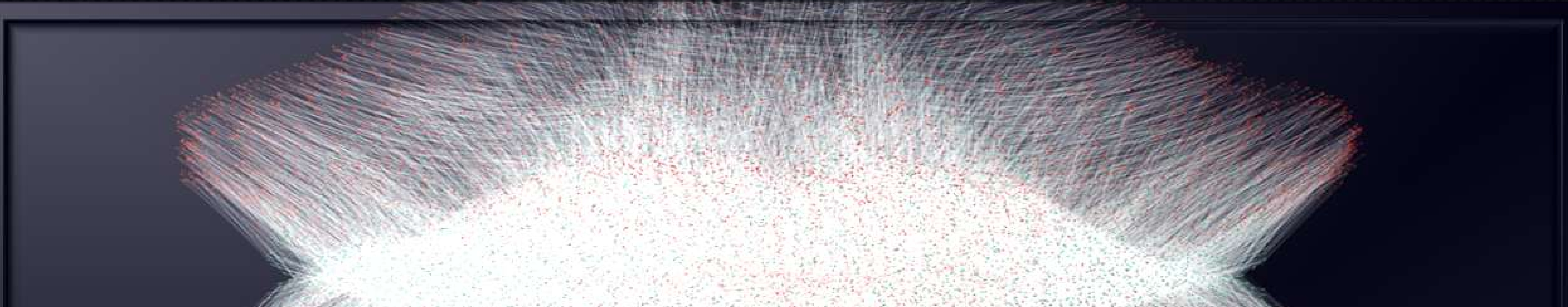
- Rev supports the standard SST::Statistics telemetry methods
- Each instance of Rev collects unique statistics for instruction completion, memory events and network events
- Telemetry data is output using the standard SST paths
- We have performed experiments with 500K+ unique telemetry data points with minimal simulation overhead



```
cpu:memory, ReadBytes, , Accumulator, 6077000, 0, 0, 0, 0, 0, 0
cpu:memory, WriteInFlight, , Accumulator, 6077000, 0, 347, 347, 347, 1, 1
cpu:memory, WritePending, , Accumulator, 6077000, 0, 347, 347, 347, 1, 1
cpu:memory, WriteBytes, , Accumulator, 6077000, 0, 0, 0, 0, 0, 0
cpu:memory, FlushInFlight, , Accumulator, 6077000, 0, 0, 0, 0, 0, 0
cpu:memory, FlushPending, , Accumulator, 6077000, 0, 0, 0, 0, 0, 0
cpu:memory, ReadLockInFlight, , Accumulator, 6077000, 0, 0, 0, 0, 0, 0
cpu:memory, ReadLockPending, , Accumulator, 6077000, 0, 0, 0, 0, 0, 0
cpu:memory, ReadLockBytes, , Accumulator, 6077000, 0, 0, 0, 0, 0, 0
cpu:memory, WriteUnlockInFlight, , Accumulator, 6077000, 0, 0, 0, 0, 0, 0
cpu:memory, WriteUnlockPending, , Accumulator, 6077000, 0, 0, 0, 0, 0, 0
cpu:memory, WriteUnlockBytes, , Accumulator, 6077000, 0, 0, 0, 0, 0, 0
cpu:memory, LoadLinkInFlight, , Accumulator, 6077000, 0, 0, 0, 0, 0, 0
cpu:memory, LoadLinkPending, , Accumulator, 6077000, 0, 0, 0, 0, 0, 0
cpu:memory, StoreCondInFlight, , Accumulator, 6077000, 0, 0, 0, 0, 0, 0
cpu:memory, StoreCondPending, , Accumulator, 6077000, 0, 0, 0, 0, 0, 0
cpu:memory, CustomInFlight, , Accumulator, 6077000, 0, 0, 0, 0, 0, 0
cpu:memory, CustomPending, , Accumulator, 6077000, 0, 0, 0, 0, 0, 0
cpu:memory, CustomBytes, , Accumulator, 6077000, 0, 0, 0, 0, 0, 0
cpu:memory, FencePending, , Accumulator, 6077000, 0, 460, 460, 460, 1, 1
cpu:memory, AMOAddBytes, , Accumulator, 6077000, 0, 0, 0, 0, 0, 0
cpu:memory, AMOAddPending, , Accumulator, 6077000, 0, 0, 0, 0, 0, 0
cpu:memory, AMOXorBytes, , Accumulator, 6077000, 0, 0, 0, 0, 0, 0
cpu:memory, AMOXorPending, , Accumulator, 6077000, 0, 0, 0, 0, 0, 0
cpu:memory, AMOAndBytes, , Accumulator, 6077000, 0, 0, 0, 0, 0, 0
cpu:memory, AMOAndPending, , Accumulator, 6077000, 0, 0, 0, 0, 0, 0
```

REV CXX ARCHITECTURE OVERVIEW





SST+REV SCALABILITY

SST Scalability

- Several ongoing simulation efforts require that we push the bounds of SST scalability
- Sandia, TCL and other orgs have performed a number of independent scaling tests
- We sought to consolidate scaling tests into a simple, predictable set of simulation inputs
- Stretch goal: run simulations across large portions of leadership-class systems

SIMULATION CONFIGURATIONS

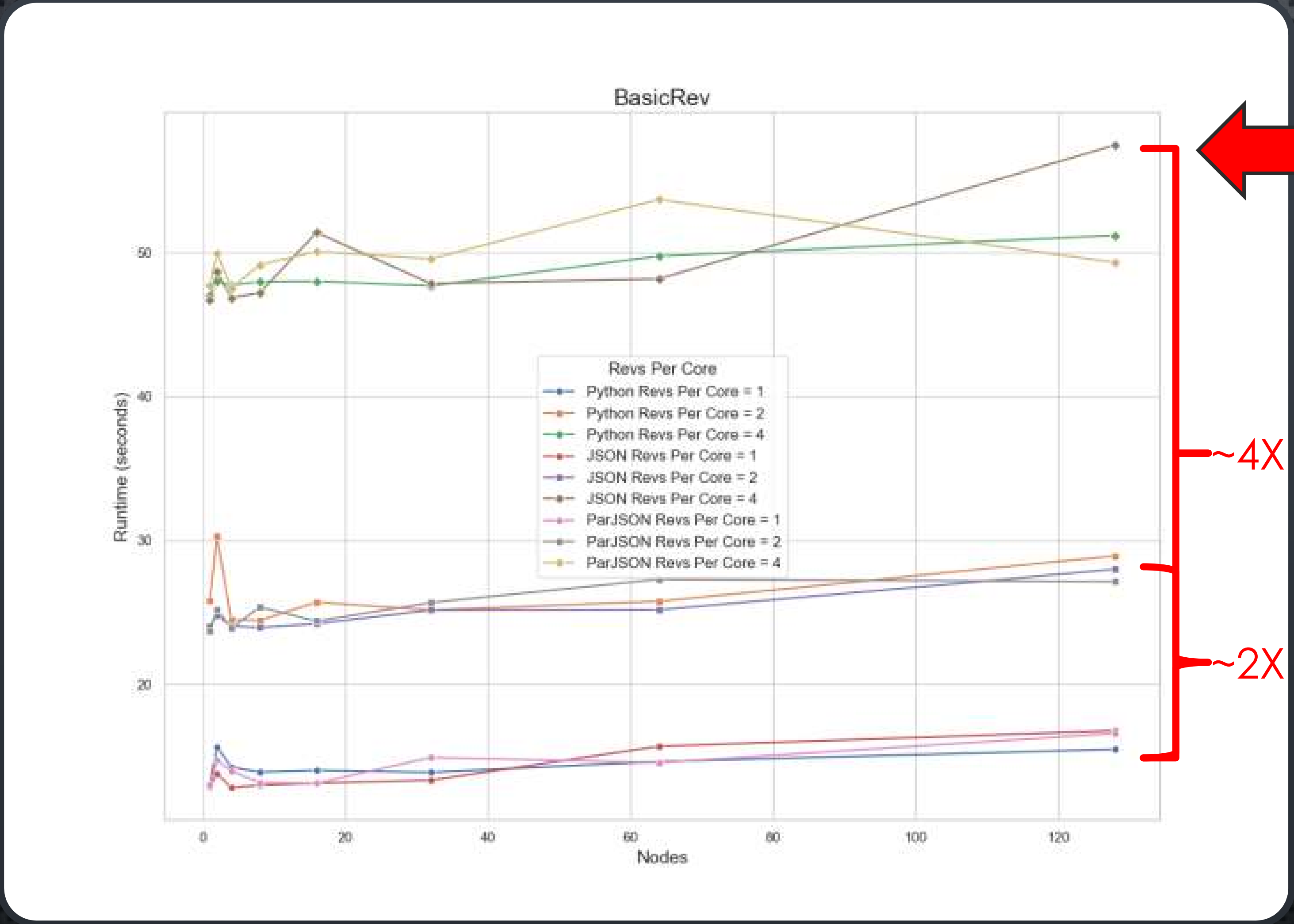
- Rev RISC-V component
  - Internal *RevMem* memory device
  - 8GB of backing memory per core
  - RV64IMAFDC
  - BIG_LOOP.exe test
  - Clocked at 2.0Ghz
- 1 component per Rev instance

- Rev RISC-V Component
  - External memHierarchy infrastructure
  - RevBasicMemCtrl @ 2.0Ghz
  - RV64IMAFDC
  - Clocked at 2.0Ghz
- memHierarchy
  - L1 Cache @ 2Ghz
    - 16KiB; LRU; MESI
  - memController @ 2.0Ghz
  - simpleMem Backend
    - 8GB of backing store
- 6 components/subcomponents per Rev instance

https://github.com/tactcomplabs/rev

Simulation Configurations cont.

1. BasicRev Python Script (uses model options to drive # of Revs)
2. BasicRev JSON (monolithic JSON file)
3. BasicRev Parallel JSON (one JSON file per MPI rank)
4. RevMemH Python Script (Uses model options to drive # of Revs)
5. RevMemH Serial JSON (monolithic JSON file)
6. RevMemH Parallel JSON (one JSON file per MPI rank)
7. ***Stats: RevMemH Python with statistics enabled



BASICREV SCALING

TAKEAWAYS

- SST+Rev provides an interesting path to simulate HPC applications at scale
- Requires some knowledge of the hardware structure in order to be successful
- Requires an existing HPC resource for large experiments

- ***SIMULATE TOMORROW'S HPC WITH TODAY'S***

TCL
0101
000110
0101010

Tactical Computing Labs

BACKUP

Scalability Factors

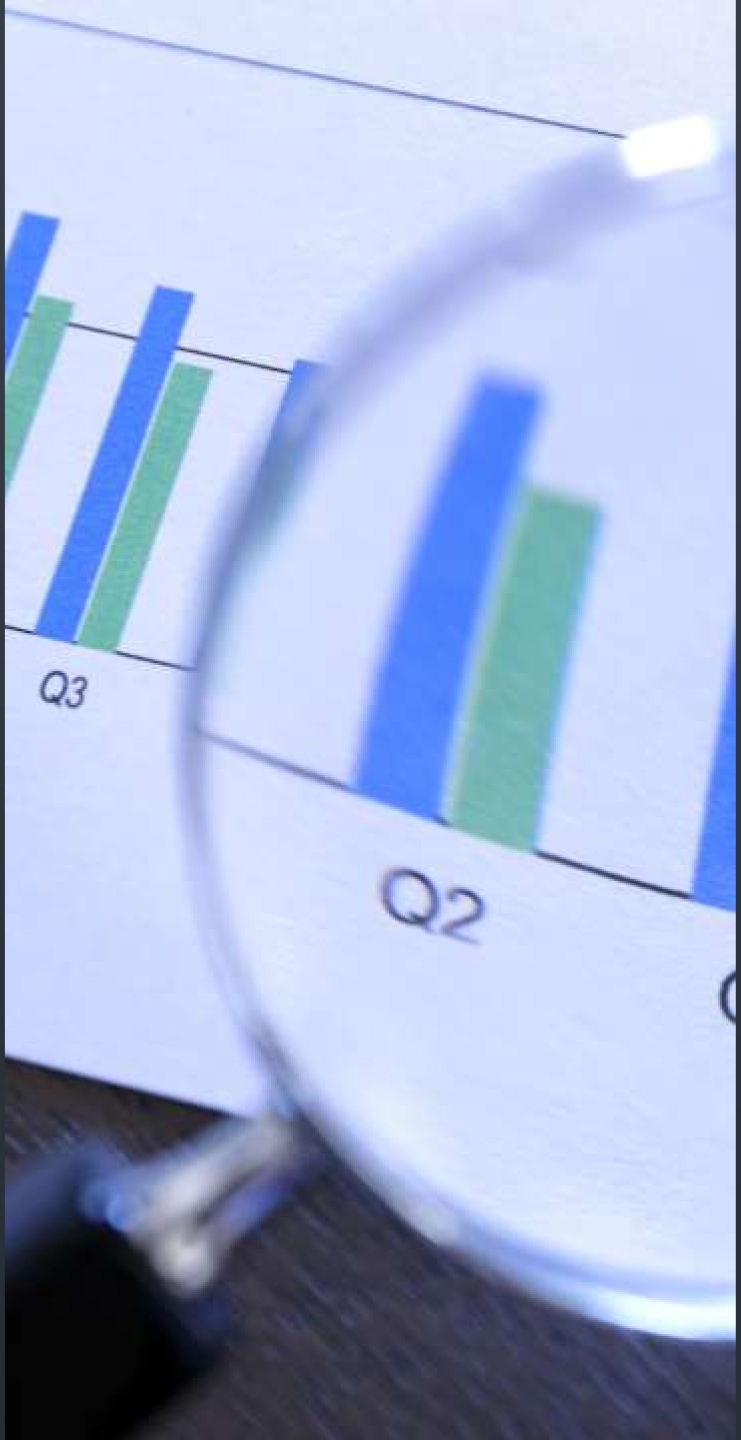
- We know there are quite a number of scalability factors:

1. Memory capacity
2. Interconnect performance (in certain cases)
3. Component configuration
4. Component oversubscription
5. Method of parallelism (MPI, Threading, Both)

PERFORMANCE TARGETS

- WE WOULD LIKE TO DEVELOP ***PREDICTABLE*** PERFORMANCE METRICS USING KNOWN GOOD SIMULATION INPUTS
- IDEAL SCALABILITY WOULD BE LINEAR, BUT WE MAY BE ABLE TO ACHIEVE SUPERLINEAR SPEEDUP ON SOME CONFIGURATIONS

- ***DEVELOP SIMULATION METHODOLOGIES TO MODEL LARGE SYSTEMS AT SCALE W/O RUNNING FOR WEEKS ON END!***



Scalability Study

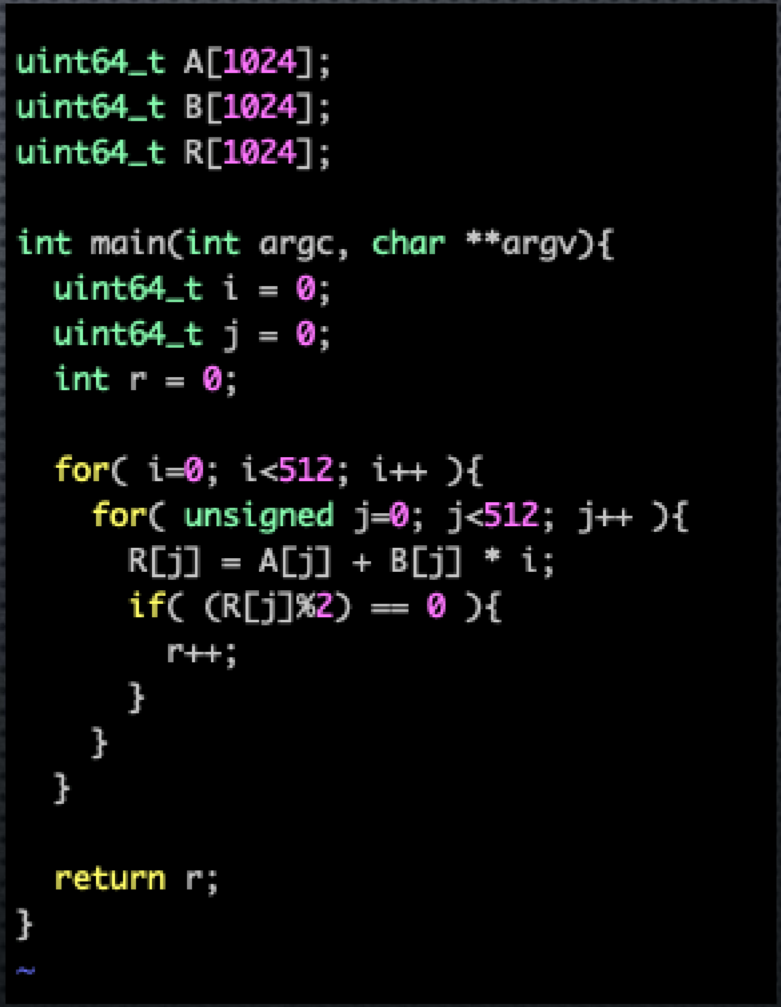
- Test the scalability of SST under controlled conditions
- Look at core scalability as a function of simulation configuration
  - ***What simulation methods should I use for a simulation of scale F(x)?***
- Deterministic event propagation
  - We don't want interconnect performance (or lack thereof) poisoning results

SIMULATION CONFIGURATIONS CONT.

1. MPI-centric (one MPI rank per core)
2. MPI+Threading (one rank per node; CXX Threads across cores)
3. MPI-Centric + Statistics
4. MPI-Ideal (distributing ***K*** components across ***K*** ranks)
5. MPI+Threading Ideal (distributing ***K*** components across ***K*** PEs)
   - where ***PEs = (N ranks * 24 threads)***

Workload

- Simple C program that executes a 2D loop structure with arithmetic
- Executes ~10,229,786 cycles @ 2.0Ghz w/ RevMem Enabled
    - ~10.2298 ms of simulated time



```
uint64_t A[1024];
uint64_t B[1024];
uint64_t R[1024];

int main(int argc, char **argv){
  uint64_t i = 0;
  uint64_t j = 0;
  int r = 0;

  for( i=0; i<512; i++ ){
    for( unsigned j=0; j<512; j++ ){
      R[j] = A[j] + B[j] * i;
      if( (R[j]%2) == 0 ){
        r++;
      }
    }
  }

  return r;
}
~
```

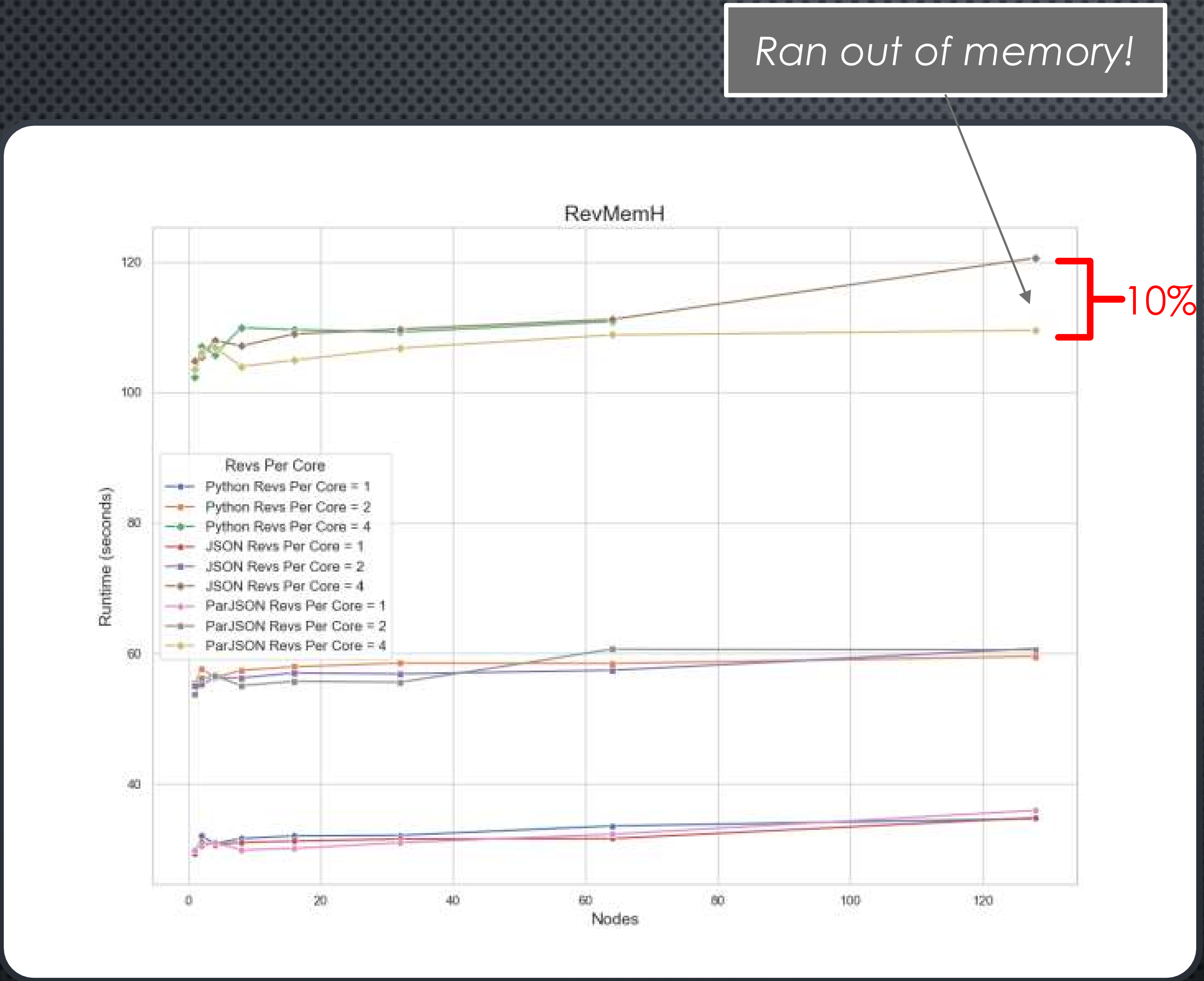
https://github.com/tactcomplabs/rev/tree/devel/test/big_loop

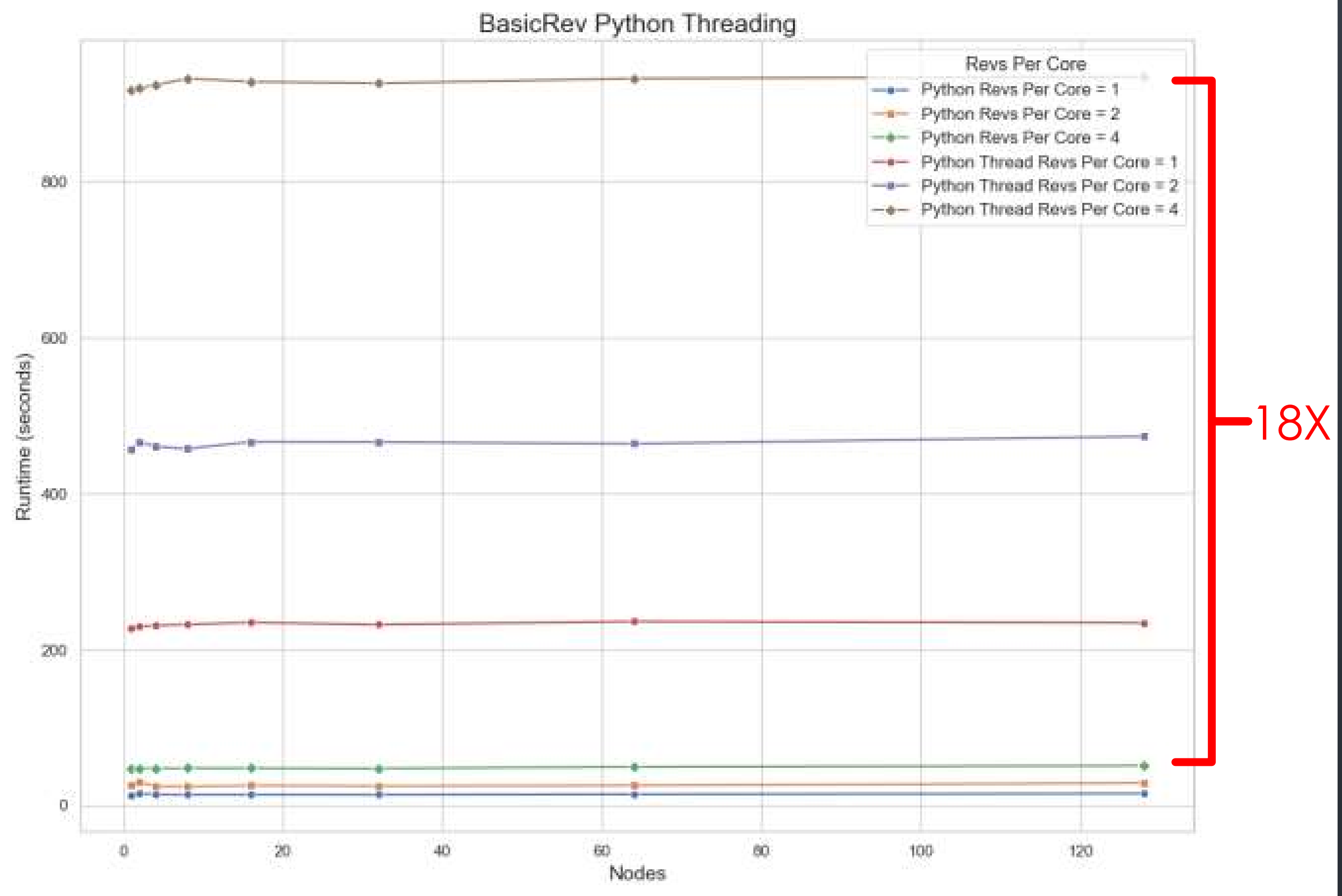
SST Configuration

- SST 13.0.0 Release w/ SST Elements 13.0.0
- Rev *DEVEL* branch @ commit 91c76abc9f197454d1b774773a5a86fde54318c1
- GCC 10.3.0
- RISC-V GNU Toolchain 12.2.0 (newlib+ELF)
- OpenMPI 4.1.4
- Python 3.9.12

- Georgia Tech **PACE** HPC infrastructure
- CPU-192GB Configuration:
  - Dual intel Xeon Gold 6226 @ 2.7Ghz (24 cores per node)
  - 192GB DDR4 memory
  - Infiniband 100HDR interconnect

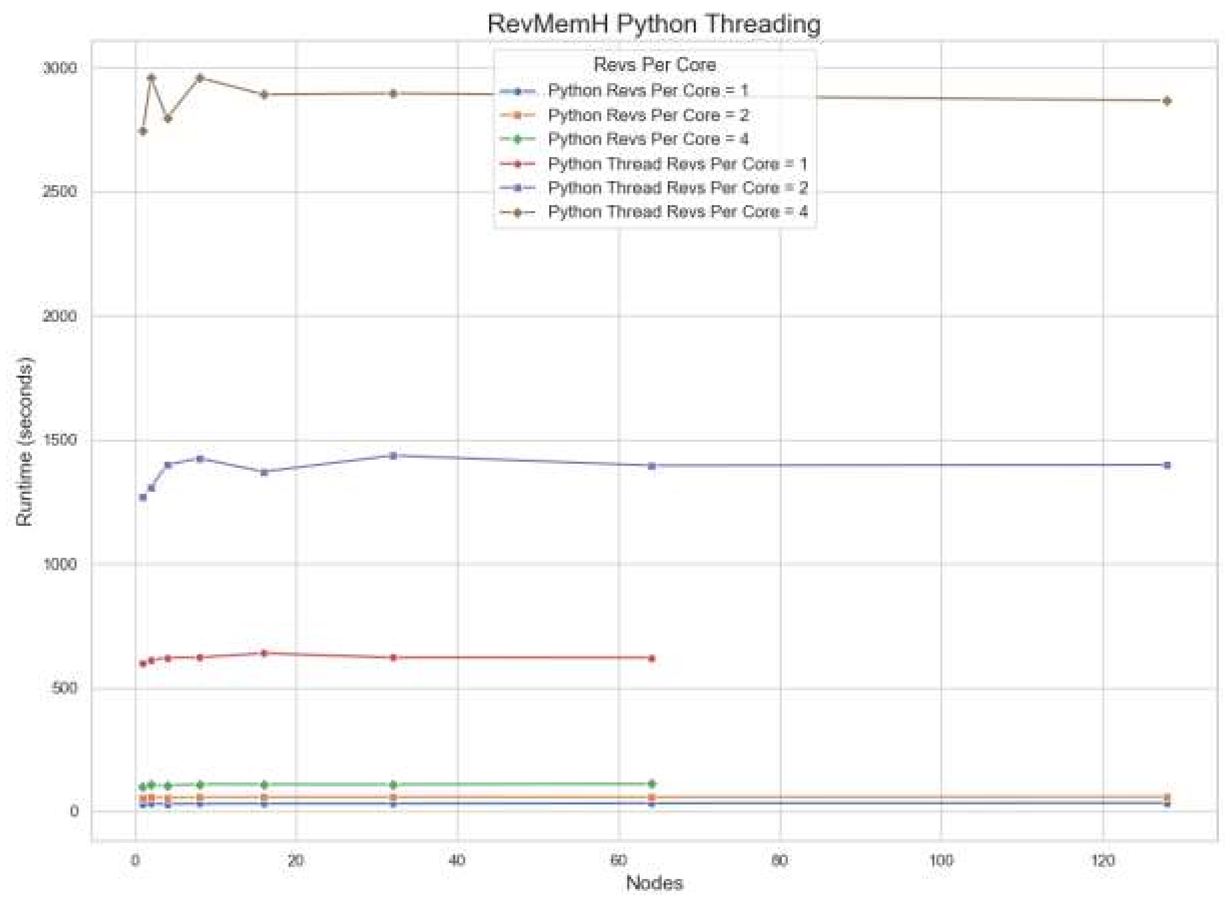
REVMEMH SCALING





BASICREV.PYTHON.THREADING

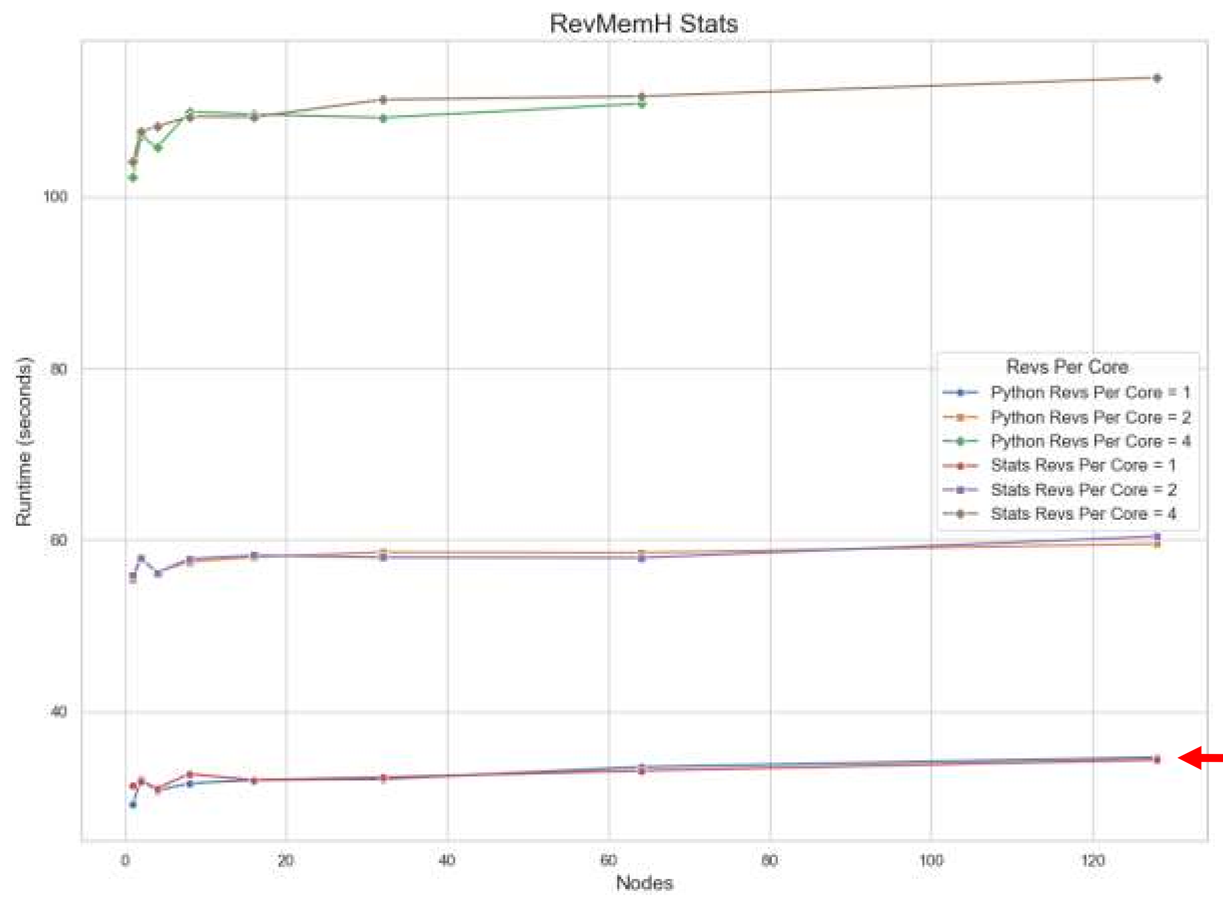
~17.7X degradation across all tests



REVMEMH.PYTHON.THREADING

~8X degradation across all tests

REVMEMH.STATS



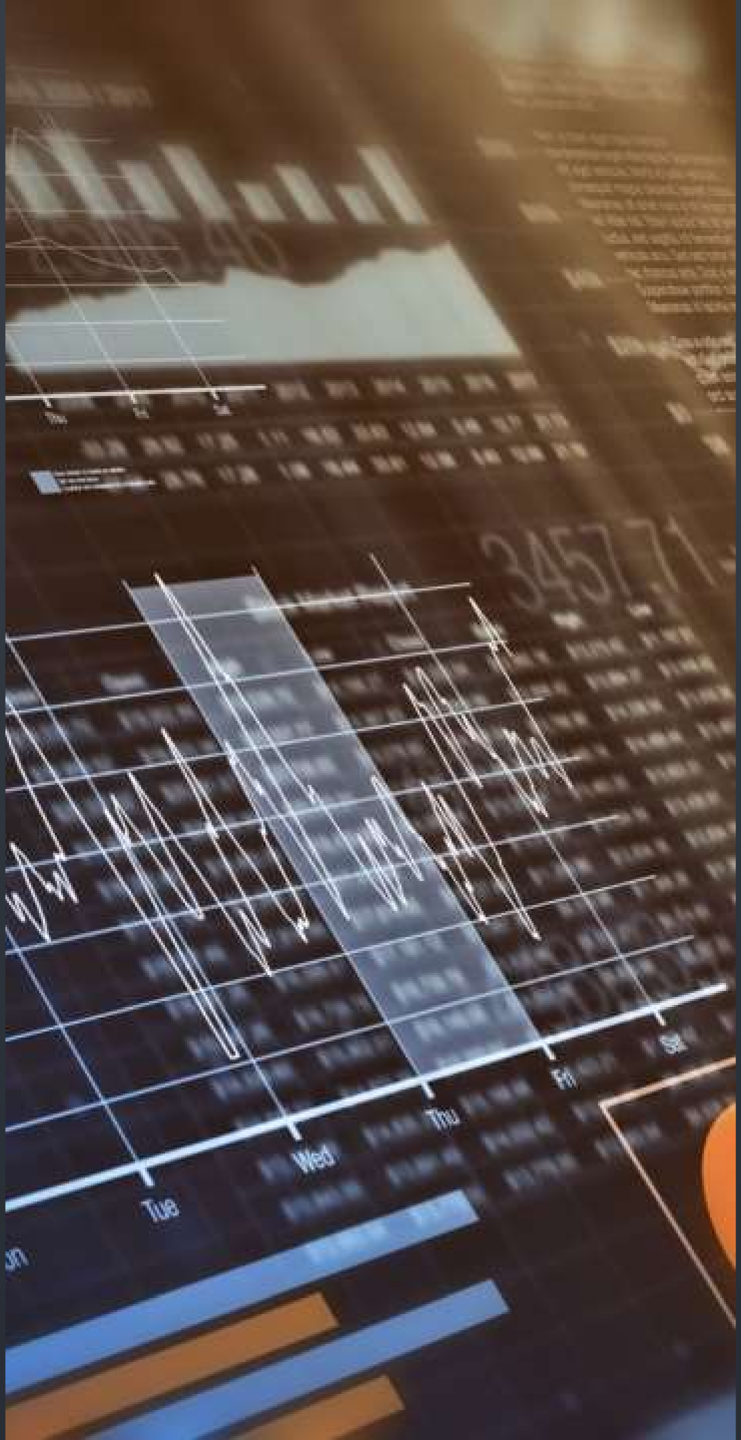
Performance is parity!

Takeaways

- SST 13.0.0 exhibits consistent scalability!
- Oversubscribing individual ranks/threads has predictable performance reductions
- Standard Python SDL's provide excellent performance baselines
- JSON Performance is reasonable, but only truly advantageous for parallel JSON on large number of MPI ranks
- Statistics have almost no performance degradation

Simulation Tuning

- Scalability is quite good using the three modeled simulation input types
  - Choose your SDL input based upon the relative scale of the simulation
- Pay attention to memory capacity!
  - Oversubscription per rank provides an easy path to larger simulations
  - Depending upon the target components, may require large amounts of host memory
- MPI currently provides the best performance at all scales
- **Note:** We did not experiment with manually partitioned graphs!
  - The default partitioner appears to be sufficient



Simulation Statistics

- Simulation statistics have little to no performance degradation!
- Rev alone has 40+ statistics
- memHierarchy layers add a tremendous number of statistics values
- **Result:** Add statistics and enable statistics at will. The only performance issue is related to downstream Analysis