TUTORIAL OUTLINE

- Session I (9:00-9:35AM): Introduction to CODES and its use for HPC System Simulations

- Session II (9:40-10:15AM): Replaying HPC traces using TraceR

- Coffee Break (10:15-10:45AM)

- Session III (10:45-12:15 PM): Visual Analysis of CODES Network Simulations

- Hackathon
CODES TEAM

- **CODES Core**: Rob Ross, Phil Carns, Misbah Mubarak, Matthieu Dorier, Shane Snyder
  - Argonne National Laboratory, Mathematics and Computer Science

- **ROSS Core**: Chris Carothers, Caitlin Ross & Noah Wolfe
  - Rensselaer Polytechnic Institute, Computer Science

- **CODES Vis**: Kwan-Liu Ma, Kelvin Li
  - UC Davis, Computer Science

- **TraceR and ECP Project**: Abhinav Bhavele, Nikhil Jain
  - Lawrence Livermore National Laboratory, Center for Applied Scientific Computing
GOALS OF SESSION I

- Introduction
- Features offered by CODES-TraceR
- Statistics collection with CODES/ROSS
- Interpreting simulation output
CODES MODELING & SIMULATION FRAMEWORK

- Accelerate HPC system co-design by providing a detailed simulation of HPC interconnects, storage, workloads and surrounding environment
- Couple best-of-breed parallel discrete event simulation with experts in interconnects and storage architecture design
- Incrementally develop HPC simulation capability, validating approach and components along the way
- Complement experimentation on real systems
Discrete event simulation (DES): a computer model for a system where changes in the state of the system occur at discrete points in simulation time.

Parallel DES allows execution of simulation on a parallel platform.

Rensselaer Optimistic Simulator System (ROSS) provides PDES capability for CODES.
  - Optimistically schedules events. Rollback realized via reverse computation.
  - Logical processes (LPs) model state of the system.
CODES-TRACER HPC SIMULATIONS

- Packet-level simulations of HPC interconnect topologies
- Trace-driven analysis, synthetic workloads, online workloads
- Multiple jobs can be replayed on the network
- Different job placement schemes can be used
- Multiple ranks mapped to network nodes can be used
- Detailed statistics generation
- MPI collective operations can be simulated
- General purpose storage model that uses concurrent, pipelined RDMA read/write requests (simulating burst buffers/SSD)
SIMULATION STACK

- DUMPI
- Comm Patterns
- OTF2
- BigSim
- Others

- MPI Simulation layer
- Synthetic workloads
- TraceR

- CODES System Models

- ROSS

- Any machine with MPI support
General framework for replaying traces on HPC interconnect simulation

- Postmortem network traces
- CODES Network workload component
- CoRTex collective translation library
- Synthetic traffic patterns
- CODES Network Models
- Send /Receive Network messages
- Feeds MPI operations

CODES specific framework for replaying traces on HPC interconnect simulations
LEVERAGING ARGOBOTS FOR HPC COMMUNICATION REPLAY

- Argobots is used to synchronize the execution of HPC skeleton applications in parallel with the CODES network simulations.
- CODES creates a separate ULT to represent each process in the skeleton application.
- Each ULT runs the skeleton code for that process, issues MPI calls that are stubs queued as events in the simulator.
- The CODES simulation thread simulates the events and yields to the skeleton code ULT when it runs out of events to process.
INSTALLATION & SETUP 1/3

- **ROSS INSTALLATION**
  - Download ROSS repo: `git clone http://github.com/carothersc/ROSS.git`
  - Configure by making a build directory: `cd build`
  - `ARCH=x86_64 CC=mpicc CXX=mpicxx cmake`
  - `-DCMAKE_INSTALL_PREFIX=../install ../`
  - `make -j 3 && make install`

- **CODES INSTALLATION**
  - Download CODES repo: `git clone https://xgitlab.cels.anl.gov/codes/codes.git`
  - `./prepare.sh`
  - Configure in build directory: `cd build`
  - `../configure --prefix=/path/to/install CC=mpicc CXX=mpicxx`
  - `PKG_CONFIG_PATH=/path/to/ross/install/lib/pkgconfig`
  - `Do both make && make tests`
INSTALLATION & SETUP 2/3

- **ARGOBOTS INSTALLATION**
  - `git clone https://github.com/pmodels/argobots.git`
  - `./autogen.sh`
  - `../configure --prefix=$prefix-dir/argobots-install`
  - `make && make install`

- **CODES installation with SWM**
  - Download CODES repo: `git clone https://xgitlab.cels.anl.gov/codes/codes.git`
  - `./prepare.sh`
  - Configure in build directory: `cd build`
  - `../configure --prefix=/path/to/install CC=mpicc CXX=mpicxx PKG_CONFIG_PATH=/path/to/ross/install/lib/pkgconfig:/path/to/argobots/pkgconfig -with-boost=/boost/path -with-swm=/swm/conf/path`
  - Do both `make && make tests`
INSTALLATION & SETUP 3/3

**DUMPI INSTALLATION**
- `git clone https://github.com/sstsimulator/sst-dumpi`
- `CFLAGS="-DMPICH_SUPPRESS_PROTOTYPES=1 -DHAVE_PRAGMA_HP_SEC_DEF=1"
- `./bootstrap.sh`
- `./configure --enable-libundumpi CC=mpicc --prefix=$INSTALL_PATH`
- Use `-with-dumpi=/path/to/dumpi/install` option to enable DUMPI with CODES

**CORTEX INSTALLATION**
- `git clone https://xgitlab.cels.anl.gov/mdorier/dumpi-cortex.git`
- `Configure in build directory: cd build`
- `cmake .. -G "Unix Makefiles" -DMPICH_FORWARD:BOOL=TRUE -DCMAKE_INSTALL_PREFIX=$HOME/CODES/install/cortex -DDUMPI_ROOT=$HOME/CODES/install/dumpi`

**For installation details see:** [https://xgitlab.cels.anl.gov/codes/codes/wikis/installation](https://xgitlab.cels.anl.gov/codes/codes/wikis/installation)
CONFIGURING NETWORK SIMULATIONS
SIMPLE NET LOGP NETWORK MODEL

- A latency/bandwidth model where message is directly sent from source to destination
- Uses infinite queuing
- Easy setup—uses a startup delay and link bandwidth for configuration
- Mostly for debugging/testing purposes-- Can be used as a starting point when replaying MPI traces
- It can be used as a baseline network model with no contention and no routing
CONFIGURING SIMPLE-NET LOGP MODEL

For mapping entities on ROSS MPI processes

Messages are broken into packets by the model-net layer

ROSS specific parameter (event size)

Startup delay in ns

Link bandwidth in MB/s (one link between each pair of nodes)

Configuration file can be found in codes/tests/conf/modelnet-test.conf
RUNNING A SIMPLE-NET LOGP MODEL

- `./tests/modelnet-test --sync=1 -- tests/conf/modelnet-test.conf`

- A simple test in which a simulated MPI rank sends message to the next rank, which replies back

- Continues until a certain number of messages is reached
DRAGONFLY NETWORK MODEL

- Network configuration files are used for setting up network connections
- Can simulate different dragonfly configurations e.g. ones used by IBM PERCS and Cray XC systems
- Multiple forms of routing are supported: minimal, adaptive, non-minimal and progressive adaptive
- Packet based simulation with credit based flow control
- Uses multiple virtual channels for deadlock prevention
CONFIGURING DRAGONFLY NETWORK MODEL

nw-lp is a simulated MPI process
A simulated dragonfly network node
A simulated dragonfly network router

- For simulating multiple MPI processes per node \( \rightarrow \) nw-lp=num-procs * number of network nodes
- Self messages \( \rightarrow \) messages sent to the same network node
- Overhead for sending self message can be configured

Configuration file can be found in codes/src/network-workloads/dragonfly-custom
Router arrangement within a group. Should match the input network configuration

Buffer size of virtual channels can be configured

Number of compute nodes per router is configurable

Network configuration files – can be custom generated (see scripts/gen-cray-topo/README.txt). Optionally, use the base dragonfly model to avoid network config files.
RUNNING A DRAGONFLY NETWORK SIMULATION

- Download the traces:
  - wget https://portal.nersc.gov/project/CAL/doe-miniapps-mpi-traces/AMG/df_AMG_n1728_dumpi.tar.gz

- Run the offline simulation:
  - ./src/network-workloads/model-net-mpi-replay --sync=1 --disable_compute=1 --workload_type="dumpi" --workload_file=df_AMG_n1728_dumpi/dumpi-2014.03.03.14.55.50- --num_net_traces=1728 -- ../src/network-workloads/conf/dragonfly-custom/modelnet-test-dragonfly-edison.conf

- Run the online simulation:
VIRTUAL TIME-STAMP SAMPLING
ROSS INSTRUMENTATION WITH CODES


- Models that have instrumentation support implemented:
  - nw-lp (model-net-mpi-replay.c)
  - dfly server LP (model-net-synthetic.c)
  - custom dfly server LP (model-net-synthetic-custom-dfly.c)
  - fat tree server LP (model-net-synthetic-fattree.c)
  - slimfly server LP (model-net-synthetic-slimfly.c)
  - original dragonfly router and terminal LPs (dragonfly.c)
  - dragonfly custom router and terminal LPs (dragonfly-custom.C)
  - slimfly router and terminal LPs (slimfly.c)
  - fat tree switch and terminal LPs (fat-tree.c)
  - model-net-base-lp (model-net-lp.c)
CALLBACK FUNCTIONS – VIRTUAL TIME SAMPLING

• Callback signature similar to a normal ROSS event
  • Provides a void * (here is cast to struct dfly_cn_sample*)
    • This is a pointer provided by ROSS to save your data to the instrumentation data buffer
  • Reverse function is only necessary if you change LP state at all
  • No message pointer
    • The LP actually calling this function is the specialized ROSS LP, so there is no normal model event associated with this sampling event
• Other sampling methods similar, except no reverse handlers
CALLBACK FUNCTIONS – EVENT TRACING

- Can access the event/message data
  - Can use the same function for multiple LP types if desired
  - But they need to have the same message type (or you need to appropriately cast when dereferencing)
- Can access LP (and through that, the LP state)
  - \((\text{lp}_\text{state}_\text{type}*)\text{lp}->\text{cur}_\text{state}\)
- Buffer pointer provided by ROSS for you to copy the data
- Collect\_flag allows you to turn on/off tracing this particularly event
  - By default, ROSS sets the value to 1 (no need to do anything if you want to always collect the event data)
  - Set to 0 if you don’t want to collect a particular event in some situation
CALLBACK FUNCTION REGISTRATION

Event Tracing callback function
GVT-based and Real time sampling callback function
Virtual Time sampling callback function (forward)
Virtual Time sampling callback function (reverse)
These functions register the instrumentation callbacks with CODES, so it can do the appropriate registration with ROSS
FOUR STEPS TO PERFORMANCE PREDICTIONS

1. Prototype system design
   – Discussed in the previous session
   – Set up using network parameters

2. Workload selection
   – Depends on the use case
   – Application traces
   – Synthetic patterns
   – Skeletons

3. Workload creation

4. Simulation
WORKLOADS

- **Synthetic Workloads:**
  - Follow specific communication pattern and a constant injection rate
  - Often used to stress the network topology to identify best and worst case performance
  - Examples include uniform random, all to all, bisection pairing, bit permutation
  - Don’t require simulation of MPI operations

- **HPC Application Traces:**
  - Useful for network performance prediction of production HPC applications
  - Trace size can be large for long running or communication intensive applications
  - Potential to capture computation-communication interplay
  - Require accurate simulation of MPI operations
  - Simulation results can be complex to analyze
DUMPI MPI TRACE LIBRARY

- Provides trace collection and replay tools for MPI based applications
- Trace collection is simple – link the MPI application with libdumpi
- Trace can be replayed using libundumpi utility
- Libundumpi provides callbacks you can use when MPI operations are replayed
- Preserves the causality order of MPI operations
- Captures detailed statistics for each MPI operation call
Repository can be cloned at:
   – *git clone https://github.com/sstsimulator/sst-dumpi.git*

Configure and build using any MPI compiler

Make sure to use ‘—enable-libdumpi’ when configuring

Once installed, simply add ‘-L$(DUMPI_INSTALL) -ldumpi’ in your application

DUMPI traces will be generated automatically with each application run

Naming convention: dumpi-yyyy.dd.mm.hh.mm.ss-MPI-RANK-ID.bin

More information can be found at: *https://github.com/sstsimulator/sst-dumpi*

HPC application traces in DUMPI format: *https://portal.nersc.gov/project/CAL/designforward.htm*
GENERATING OTF2 TRACES (1/2)

- New Open Trace Format version 2 is supported by several tools
- ScoreP - Scalable Performance Measurement Infrastructure for Parallel Codes
- Tool suite with several libraries and helper tools
- Inside ScoreP source directory
  - `CC=mpicc CFLAGS="-O2" CXX=mpicxx CXXFLAGS="-O2" FC=mpif90 F77=mpif77 ./configure --without-gui --prefix=<SCOREP_INSTALL>
  - `make && make install`
  - Make sure ScoreP installation’s bin directory is in PATH
- Simple case: change the application linker to
  `LD = scorep --user --nocompiler --noopenmp --nopomp --nocuda --noopenacc --noopencl --nomemory <your_linker>`
GENERATING OTF2 TRACES (2/2)

- Before running, set the following environment variables:
  
  ```
  export SCOREP_ENABLE_TRACING=1
  export SCOREP_ENABLE_PROFILING=0
  export SCOREP_MPI_ENABLE_GROUPS=ENV,P2P,COLL,XNONBLOCK
  ```

- Turning tracing on/off: make sure these calls are synchronized
  
  ```
  #include <scorep/SCOREP_User.h>
  SCOREP_RECORDING_ON(); - start recording
  SCOREP_RECORDING_OFF(); - stop recording
  ```

- During compilation, add flags:
  
  ```
  -I$SCOREP_INSTALL/include -I$SCOREP_INSTALL/include/scorep -DSCOREP_USER_ENABLE
  ```

- Trace target options
  
  ```
  export SCOREP_TOTAL_MEMORY=256M
  export SCOREP_EXPERIMENT_DIRECTORY=/p/lscratchd/<username>/...
  ```
TRACING OUTPUT

- scorep-* directory generated with following content:
  scorep.cfg traces traces.def traces.otf2
- scorep.cfg is human readable; can be used to verify if the environment is correctly generated
- traces.otf2 is a binary meta-file
- traces is a directory that contains the details
- Use otf2-print utility in ScoreP bin to view the traces:
  otf2-print –L 0 traces.otf2
INFORMATION CAPTURED IN A TYPICAL TRACE (E.G. IN DUMPI, OTF2)

<table>
<thead>
<tr>
<th>Time stamp, $t$ (rounded off)</th>
<th>Operation type</th>
<th>Operation data (only critical information is highlighted)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t = 10$</td>
<td>MPI_Bcast</td>
<td>root, size of bcast, communicator</td>
</tr>
<tr>
<td>$t = 10.5$</td>
<td>MPI_Irecv</td>
<td>source, tag, communicator, req ID</td>
</tr>
<tr>
<td>$t = 10.51$</td>
<td>user_computation</td>
<td>optional region name - “boundary updates”</td>
</tr>
<tr>
<td>$t = 12.51$</td>
<td>MPI_Isend</td>
<td>dest, tag, communicator, req ID</td>
</tr>
<tr>
<td>$t = 12.53$</td>
<td>user_computation</td>
<td>optional region name - “core updates”</td>
</tr>
<tr>
<td>$t = 22.53$</td>
<td>MPI_Waitall</td>
<td>req IDs</td>
</tr>
<tr>
<td>$t = 25$</td>
<td>MPI_Barrier</td>
<td>communicator</td>
</tr>
</tbody>
</table>
**EXAMPLE TO SHOW THE EFFECT OF REPLAYING TRACES**

<table>
<thead>
<tr>
<th>Original Time stamps</th>
<th>Original duration</th>
<th>New Time stamps</th>
<th>New duration</th>
<th>Operation type</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.5</td>
<td>10</td>
<td>0.2</td>
<td>MPI_Bcast</td>
</tr>
<tr>
<td>10.5</td>
<td>0.01</td>
<td>10.2</td>
<td>0.01</td>
<td>MPI_Irecv</td>
</tr>
<tr>
<td>10.51</td>
<td>2</td>
<td>10.21</td>
<td>2</td>
<td>user_computation</td>
</tr>
<tr>
<td>12.51</td>
<td>0.02</td>
<td>12.21</td>
<td>0.02</td>
<td>MPI_Isend</td>
</tr>
<tr>
<td>12.53</td>
<td>10</td>
<td>12.23</td>
<td>10</td>
<td>user_computation</td>
</tr>
<tr>
<td>22.53</td>
<td>2.47</td>
<td>22.23</td>
<td>0.03</td>
<td>MPI_Waitall</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>22.26</td>
<td>1.7</td>
<td>MPI_Barrier</td>
</tr>
</tbody>
</table>
DUMPI VS OTF2

- Most of the information in the trace format is the same
- Different w.r.t. capturing of dynamically determined events: e.g. MPI_Waitany

DUMPI: stores all the information passed to the MPI call
- Simulation decides which request to fulfill: accurate resolution for target systems
- If the control flow of the program can change significantly due to the ordering of operations, simulations are not entirely correct

OTF2: stores only the information that is used (e.g. which request was satisfied)
- Accurately mimics the control flow of the trace run
- But does not accurately represent execution for the target system

Shortcoming of trace based simulations!
SIMULATING MPI
MPI SIMULATION

- Matching semantics and standard has to be followed for a correct simulation
  - So obviously done
- Eager – Rendezvous protocol
  - Cutoff can be specified in the config
- Library call overheads handled using a constant cost
- Collectives:
  - OTF2 based simulations implements them internally
  - DUMPI based simulations use Cortex
TRANSLATING MPI CALLS USING CORTEX

- Intended to translate MPI calls (for e.g. collectives) into a set of another MPI calls (e.g. point to point)
- Cortex comes with a set of translation functions to convert collectives into point to point using MPICH algorithms
- When linked with DUMPI and CODES, Cortex translates MPI collectives into point to point sends/receives (simulated by CODES)
- Cortex can also be used to implement your own translation functions (e.g. collective algorithms) --- *Will discuss it more in Session III*
- Cortex tutorial is available at: https://xgitlab.cels.anl.gov/mdorier/dumpi-cortex/wikis/home
CODES, CORTEX AND DUMPI INTERACTION

DUMPI Application Trace

Cortex

MPI send/recv/waits

MPI Collective calls

Translated sends/recvs

CODES

MPI Simulation Layer

Model-net layer

Network Models
**MPI TRANSLATION WITH CORTEX**

- To enable collective translation, install Cortex and reconfigure CODES with Cortex

- Cortex available for download: git clone https://xgitlab.cels.anl.gov/mdorier/dumpi-cortex.git

- `cmake .. -G "Unix Makefiles" -DMPICH_FORWARD:BOOL=TRUE -DCMAKE_INSTALL_PREFIX=$HOME/CODES/install/cortex -DDUMPI_ROOT=$HOME/CODES/install/dumpi`

- See instructions at: https://xgitlab.cels.anl.gov/codes/codes/wikis/codes-cortex-install

- Use `–with-cortex=/path/to/cortex/install` option
IN A NUTSHELL: REPLAYING A SINGLE APPLICATION TRACE

```
./bin/model-net-mpi-replay --sync=1 --disable_compute=1 --workload_type="dumpi" --workload_file=dumpi-2014.03.03.14.55.50- --num_net_traces=1728 -- modelnet-test-dragonfly-edison.conf
```

- Runtime options
  - `--num_net_traces`: Number of input network traces
  - `--workload_file`: DUMPI trace file
  - Network configuration file: Any of the network files (number of simulated ranks > number of ranks in trace)
  - `--lp-io-dir` (optional): Generates detailed network counters and statistics
  - `--lp-io-use-suffix` (optional): Generates a unique directory per run
  - `--disable_compute` (optional): disable any compute time between MPI events

- For running parallel simulations, use `--sync=3`
- By default, linear job mapping is used
THANK-YOU!
STORAGE MODELS AND SYNTHETIC TRAFFIC GENERATION
GOALS OF THE SESSION

- How to do storage placement on networks?
- How to generate background network traffic?
- Using model-net API
- PDES and Networks Internal
- Continue with hands on exercises
STORAGE PLACEMENT ON INTERCONNECTS
CODES STORAGE MODEL

- General purpose model for read and write operations
- Concurrent, pipelined RDMA requests
- Comprises of the following:
  - a storage manager
  - a disk/local storage model
  - A resource tracker
- Placement of storage over the network can be modified using the network config file
PROTOCOL FOR WRITE OPERATIONS

1. Write Request
2. Reserve Disk Space (Blocking)
3. Send Response
4. Pull Data
5. Write Data
USING THE STORAGE MODEL

- codes_store_init_req (is_write, priority, obj_id, xfer_offset, xfer_size, codes_req) → For initializing the request
- codes_store_send_req(codes_req, dest_id, sender, network_id, mapping_context, ..) → For sending the request
- codes_store_send_req_rc → For reverse computation

Repo available at:
https://xgitlab.cels.anl.gov/codes/codes-storage-server
CONFIGURING STORAGE OVER THE NETWORK

- Number of concurrent requests
- Buffer size for each thread
- Size of the Memory (RAM)
- Storage size (for disk/LSM)
- Aggregate memory+storage size
- Disk bandwidth/seek configuration
CONFIGURING STORAGE OVER THE NETWORK

Two storage manager entities per 60 clients/compute nodes

Local storage model entity (disk). One to one correspondence

A total of 64 network nodes

If the data from burst buffer needs to be drained to the external storage entity
GENERATING BACKGROUND NETWORK TRAFFIC
WHY BACKGROUND TRAFFIC?

- On production HPC systems, a significant fraction of network nodes can be occupied
- How to introduce communication interference if a single application trace is being replayed on the simulation?
- Running multiple traces at a large-scale can be expensive
- One solution is to mix synthetic traffic patterns and HPC application traces
EXAMPLE SYNTHETIC PATTERNS

- Uniform Random: A network node is equally likely to send to any other network node (traffic distributed throughout the network)
- All to All: Each network node communicates with all other network nodes
- Nearest neighbor: A network node communicates with near by network nodes (or the ones that are at minimal number of hops)
- Permutation traffic: Source node sends all traffic to a single destination based on a permutation matrix
- Bisection pairing: Node 0 communicates with Node ‘n’, node 1 with ‘n-1’ and so on.
- …
SYNTHETIC TRAFFIC IN CODES

/* in case of uniform random traffic, send to a random destination. */
if (traffic == UNIFORM)
{
  b->c1 = 1;
  local_dest = tw_rand_integer(lp->rng, 0, num_nodes - 1);
}
else if (traffic == NEAREST_GROUP)
{
  local_dest = (local_id + num_nodes_per_grp) % num_nodes;
  //printf("\n LP %ld sending to %ld num nodes %ld", local_id, local_dest, num_nodes);
}
else if (traffic == NEAREST_NEIGHBOR)
{
  local_dest = (local_id + 1) % num_nodes;
  // printf("\n LP %ld sending to %ld num nodes %ld", rep_id * 2 + offset, local_dest, num_nodes);
}
assert(local_dest < num_nodes);
// codes_mapping_get_lp_id(group_name, lp_type_name, anno, 1, local_dest / num_servers_per_rep, local_dest % num_servers_per_rep, &global_dest);
global_dest = codes_mapping_get_lpid_from_relative(local_dest, group_name, lp_type_name, NULL, 0);
ns->msg_sent_count++;  
model_net_event(net_id, "test", global_dest, PAYLOAD_SZ, 0.0, sizeof(svr_msg), (const void*)m_remote, sizeof(svr_msg), (const void*)m_local, lp);

- Typical patterns supported are uniform random and nearest neighbor.
- All to all and stencil patterns have been tested (pending integration)
- See `src/network-workloads/model-net-synthetic-custom-dfly.c` and related files
GENERATING BACKGROUND TRAFFIC WITH CODES

- Communication based on uniform random traffic
- Kicks off when the main workload starts
- A notification is sent to the background traffic node to stop generating traffic once the main workload finishes
- How to enable synthetic traffic generation?
- Simply add “synthetic” instead of DUMPI trace path in workloads config file

```
216 synthetic
125 /path/to/Multigrid/Multigrid_125/dumpi-2014.03.06.23.48.13-
```
USING MODEL-NET API
MODEL NET – An abstraction layer on top of network models – topology details are specified through the config files

A valid network configuration file – examples can be found in the repo

Network model must be registered – `model_net_register`

CODES mapping must be setup – `codes_mapping_setup`

Use model-net function calls – `model_net_event(network id, source, destination, message size,...)`

Example of using model-net – `tests/model-net-test.c`
PDES AND NETWORK INTERNALS
DISCRETE EVENT SIMULATION (DES)

- Computer model for a system where changes in the state of the system occur at discrete points in simulation time
- In this model, each component of the system being simulated is represented independently via their
  - State variables
  - Virtual time
  - Events - scheduled on it and by it
Event scheduling from one component to another progresses and coordinates virtual time across components. Each plane is independently represented as a component, so is the runway.
ROSS lets users define LP (logical processes) on which events can be scheduled with time stamps.

- Each LP can have a local state that is accessible and modified only when events are executed on it.
ROSS LP

tw_lptype model_lps[] = {

    {
        (init_f) model_init,
        (event_f) model_event,
        (revent_f) model_event_reverse,
        (final_f) model_final,
        (map_f) model_map,
        sizeof(state)
    },
};
EXAMPLE OF AN EVENT FUNCTION

- Typical events act based on “type” and “content” of the message

```c
static void svr_event(
    svr_state * ns,
    tw_bf * b,
    svr_msg * m,
    tw_lp * lp)
{
    (void)b;
    switch (m->svr_event_type)
    {
    case REQ:
        handle_req_event(ns, m, lp);
        break;
    case ACK:
        handle_ack_event(ns, m, lp);
        break;
    case KICKOFF:
        handle_kickoff_event(ns, m, lp);
        break;
    case LOCAL:
        handle_local_event(ns);
```
static void handle_req_event(
    svr_state * ns,
    svr_msg * m,
    tw_lp * lp)
{
    assert(!do_pull);
    svr_msg * m_local = malloc(sizeof(svr_msg));
    svr_msg * m_remote = malloc(sizeof(svr_msg));

    m_local->svr_event_type = LOCAL;
    m_local->src = lp->gid;

    memcpy(m_remote, m_local, sizeof(svr_msg));
    m_remote->svr_event_type = ACK;

    ns->msg_recvd_count++;

    m->ret = model_net_event(net_id, "test", m->src, PAYLOAD_SZ,
        0.0, sizeof(svr_msg), (const void*)m_remote,
        sizeof(svr_msg), (const void*)m_local, lp);

    return;
}
ROSS'S LAW OF EXECUTION: FOR EVERY FORWARD ACTION, YOU MUST TELL ROSS HOW TO GO BACKWARDS

```
static void svr_event(
    svr_state * ns,
    tw_bf * b,
    svr_msg * m,
    tw_lp * lp)
{
    (void)b;
    switch (m->svr_event_type)
    {
    case REQ:
        handle_req_event(ns, m, lp);
        break;
    case ACK:
        handle_ack_event(ns, m, lp);
        break;
    case KICKOFF:
        handle_kickoff_event(ns, m, lp);
        break;
    case LOCAL:
        handle_local_event(ns);
    }
}
```

```
static void svr_rev_event(
    svr_state * ns,
    tw_bf * b,
    svr_msg * m,
    tw_lp * lp)
{
    (void)b;
    switch (m->svr_event_type)
    {
    case REQ:
        handle_req_rev_event(ns, m, lp);
        break;
    case ACK:
        handle_ack_rev_event(ns, m, lp);
        break;
    case KICKOFF:
        handle_kickoff_rev_event(ns, m, lp);
        break;
    case LOCAL:
        handle_local_rev_event(ns);
        break;
    }
```
static void handle_req_event(
    svr_state * ns,
    svr_msg * m,
    tw_lp * lp)
{
    assert(!do_pull);
    svr_msg * m_local = malloc(sizeof(svr_msg));
    svr_msg * m_remote = malloc(sizeof(svr_msg));

    m_local->svr_event_type = LOCAL;
    m_local->src = lp->gid;

    memcpy(m_remote, m_local, sizeof(svr_msg));
    m_remote->svr_event_type = ACK;

    ns->msg_recvd_count++;

    m->ret = model_net_event(net_id, "test", m->src, PAYLOAD_SZ,
                            0.0, sizeof(svr_msg), (const void*)m_remote,
                            sizeof(svr_msg), (const void*)m_local, lp);

    return;
}
APPLICATION SIMULATION IN CODES

PE 0 (mpi rank or end points)
— computation tasks, communication logs, or algorithmic state
— expected messages
— pending messages
— progress overheads

task complete

message send

message arrive

NIC
— messages to be transmitted
— packetization status of messages being transmitted
— data on buffers connecting to router
— NIC delay
— bandwidth to routers

packet send
ack/token send

packet arrive
ack/token send

packet arrive
ack/token arrive

Routers
— routing tables
— data on buffers connecting to other routers and NIC
— congestion control scheme
— pending packets in each buffer
— link bandwidth
— router delays
AVAILABLE MODELS, FEATURES, AND ADDING A NEW NETWORK MODELS

- Available: simple alpha-beta model, torus, dragonfly-(custom), fat-tree, express-mesh/hyperX
- Typical model consist of NIC (terminals) and switches/routers
- NICs
  - Common code available for within-node, message ordering, etc
  - Plugin code for individual network
- Switch/router
  - Entirely within a network model
- But, a significant fraction of node is similar for NIC plugin and switch!
NIC COMMON

- Types of queues:
  - fifo
  - round-robin
  - priority

- Other params
  - intra_bandwidth (10)
  - node_copy_queues (4)

Model_net send

delegate message to NIC size, destination,

Node facing NIC

enqueue to a common queue

Poll common queue

Going off node?

YES

NO

enqueue at destination rank at the right time

Schedule next poll event
struct model_net_method torus_method =
{
    .mn_configure = torus_configure,
    .mn_register = NULL,
    .model_net_method_packet_event = torus_packet_event,
    .model_net_method_packet_event_rc = torus_packet_event_rc,
    .model_net_method_recv_msg_event = NULL,
    .model_net_method_recv_msg_event_rc = NULL,
    .mn_get_lp_type = torus_get_lp_type,
    .mn_get_msg_sz = torus_get_msg_sz,
    .mn_report_stats = torus_report_stats,
    .mn_collective_call = NULL,
    .mn_collective_call_rc = NULL,
    .mn_sample_fn = NULL,
    .mn_sample_rc_fn = NULL,
    .mn_sample_init_fn = NULL,
    .mn_sample_fini_fn = NULL
};
Going off node? YES  

torus packet event

Has packets to send

Get credit from router

Has credit to send

Pick the next packet to send

send to router and schedule event for next send

Wait for packets

Wait for credit
CONTRIBUTING:

- Fork off on the gitlab or github repository
- Add new features
- Submit a pull request!

THANK-YOU
ADDITIONAL MATERIAL
MORE ON TRACER
TRACER – A LAYER FOR CONFIGURABLE REPLAY OF APPLICATION TRACES

- ScoreP - OTF2
- BigSim
- Others

- TraceR
- CODES
- Others

Capture application behavior by tracing runs on existing systems

Reproducing the execution: applications’ behavior, job placement and mapping, job scheduling, MPI/Charm++, etc.

Simulation of traffic flow on NICs and networks
DOCUMENTATION

- Distributed with TraceR source code
- README.md – getting started
- README.OTF - OTF2 installation and usage
- docs/UserWriteUp.txt – detailed workflow and usage
- utils/README – job placement and task mapping
AVAILABLE FEATURES

- BigSim traces for Charm++, AMPI
- OTF2 traces for MPI
- Default and user-defined job placement and task mapping
- MPI point-to-point semantics and protocols
- Inbuilt collectives: tree based bcast, reduce, allreduce, and barrier; message size based algorithms for alltoall and allgather
- Simulation time scaling
INSTALLING TRACER (1/4)

- Hosted on github: https://github.com/LLNL/tracer/
- git clone and follow README.md
- Download and install ROSS  
  - Last verified commit provided
- Download and install CODES  
  - Last verified commit provided
INSTALLING TRACER (2/4)

- Choose a trace format: BigSim or OTF2
- For BigSim, download Charm++
  - git clone http://charm.cs.uiuc.edu/gerrit/charm
- Assuming MPI is available, install two flavors of Charm++
  - For compiling codes for trace generation
    ./build bgampi mpi-linux-x86_64 bigemulator -O2
  - For compiling TraceR
    ./build charm++ mpi-linux-x86_64 bigemulator --with-production
INSTALLING TRACER (3/4)

- For OTF2, download ScoreP

- Inside ScoreP source directory
  - `CC=mpicc CFLAGS="-O2" CXX=mpicxx CXXFLAGS="-O2" FC=mpif90 F77=mpif77 ./configure --without-gui --prefix=<SCOREP_INSTALL>
  - make & & make install
  - Make sure ScoreP installation’s bin directory is in PATH
INSTALLING TRACER (4/4)

- In tracer/Makefile.common
- Set ROSS to ROSS’s installation directory
- Set CODES to CODES’s installation directory
- If using BigSim,
  - Set CHARMPATH
  - SELECT_TRACE = -DTRACER_BIGSIM_TRACES=1
- If using OTF2,
  - Make sure ScoreP installation’s bin directory is in PATH
  - SELECT_TRACE = -DTRACER_OTF_TRACES=1
- make: generates traceR executable
MORE ON GENERATING OTF2 TRACES

- ScoreP macros can be used to mark special regions
  - `SCOREP_USER_REGION_BY_NAME_BEGIN(regionname)
    SCOREP_USER_REGION_TYPE_COMMON
  ` - `SCOREP_USER_REGION_BY_NAME_END(regionname)`

- Printing simulation time at locations of interest:

- Region name with prefix `TRACER_WallTime_<any_name>` prints current time during simulation with tag `<any_name>`. 
MORE ON GENERATING OTF2 TRACES

- Simulation time looping:
- Region name TRACER_Loop can be used to mark beginning and ending of a code loop (currently once)

- In future, region names will be used for
  - Targeted kernel time modifications
  - Targeted message size modifications
int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    SCOREP_RECORDING_OFF();

    // initialization code
    MPI_Barrier(MPI_COMM_WORLD);
    SCOREP_RECORDING_ON();
    SCOREP_USER_REGION_BY_NAME_BEGIN("TRACER_Loop", SCOREP_USER_REGION_TYPE_COMMON);
    if(!myRank)
        SCOREP_USER_REGION_BY_NAME_BEGIN("TRACER_WallTime_Total", SCOREP_USER_REGION_TYPE_COMMON);
    startTime = MPI_Wtime();

while(iterations < MAX_ITER) {
    if(myRank == 0)
        SCOREP_USER_REGION_BY_NAME_BEGIN("TRACER_WallTime_InLoop", SCOREP_USER_REGION_TYPE_COMMON);
        // kernel and other code
    }
    SCOREP_USER_REGION_BY_NAME_END("TRACER_Loop");
    MPI_Barrier(MPI_COMM_WORLD);
    endTime = MPI_Wtime();
    if(!myRank)
        SCOREP_USER_REGION_BY_NAME_END("TRACER_WallTime_Total");
SCOREP_RECORDING_OFF();
}
RUNNING TRACER

- A typical run command:
  mpirun -np 8 ./traceR --sync=3 --nkp=16 --extramem=100000 --max-opt-lookahead=1000000 --timer-frequency=1000 –lp-io-dir=stats-dir -- torus.conf tracer_config

- In green, ROSS options
  - --nkp : how many KPs to create per PE = total LPs/\(-np\)
  - --extramem : how many ROSS messages to allocate = 100K should work for most cases
  - --max-opt-lookahead : optimistic leash = 1 millisecond is a good number
RUNNING TRACER

- A typical run command:
  `mpirun -np 8 ./traceR --sync=3 --nkp=16 --extramem=100000 --max-opt-lookahead=1000000 --timer-frequency=1000 --lp-io-dir=stats-dir -- torus.conf tracer_config`

- TraceR-CODES options
  - `--timer-frequency`: how frequently to print progress of task completion; optional, default = 5000
  - `--lp-io-dir`: where to write output stats; optional; code fails if the directory exists to avoid over-writing
  - `torus.conf`: network config file
  - `tracer_config`: TraceR config file
TRACER PARAMETER IN NETWORK FILE

- server in MODELNET_GRP
  - Number of processes associated with a switch
  - Assigned in a round-robin manner to nodes

- soft_delay in PARAMS
  - Approximate overhead of making an MPI/runtime call
  - In nanoseconds

- rdma_delay in PARAMS
  - Overhead of using RDMA call in rzv protocol, in nanoseconds

- eager_limit in PARAMS
  - Switch over point between eager and rzv protocols, in bytes

- copy_per_byte in PARAMS
  - Copy cost for a byte, in nanoseconds per byte
Format:
<global map file> or NA
#jobs
<path to job traces> <task mapping file or NA> <#ranks> <loop iterations>

Example:
  global_map.bin
  2
  traces-64/traces.otf2 job0 64 1
  traces-32/traces.otf2 job1 32 1
TRACER CONFIG FILE (2/2)

- At the end of file,
- `E <job id> scale_all <scale factor>`
  - Inverse scales computation time by the given factor
  - E.g. : `E 0 scale_all 40`
- `S <job id> <msg size> <replace by>`
  - Change the size of message
  - Under review, to be merged
SAMPLE OUTPUT

PE0 - LP_GID:0 : START SIMULATION, TASKS COUNT: 245611, FIRST TASK: 0, RUN TIME TILL NOW=70.000000 s, CURRENT SIM TIME 1.005877

[ 0 0 : time at task 0/245611 0.000000 ]
[ 0 0 : Begin TRACER_WallTime_MainLoop 0.000001 ]
[ 0 0 : time at task 100/245611 0.000663 ]
[ 0 0 : Begin TRACER_WallTime_NextTrajec 0.001175 ]
[ 0 0 : time at task 200/245611 0.003104 ]

....

....

[ 0 0 : time at task 245600/245611 1.485123 ]
[ 0 0 : End TRACER_WallTime_NextTrajec 1.485264 ]
[ 0 0 : End TRACER_WallTime_MainLoop 1.485265 ]