# Simulating MPI Rendezvous Protocol with CODES/TraceR

Srinivasan Ramesh, Allen D. Malony
University of Oregon

Nikhil Jain
Lawrence Livermore National Laboratory

# A little bit about me...

- First year Ph.D student in Computer Science at the University of Oregon
  - Advised by Prof. Allen D. Malony and Dr. Sameer Shende

- Research Interests: Performance Research, Tools for HPC + Data Science, Power Constrained Supercomputing, Computational Science

- Got involved with TraceR simulator through my Masters' research efforts

# Background & Motivation

# MPI libraries are complex softwares

Contain many layered, modular components interacting to affect performance

**Architecture of MVAPICH2 Software Family**

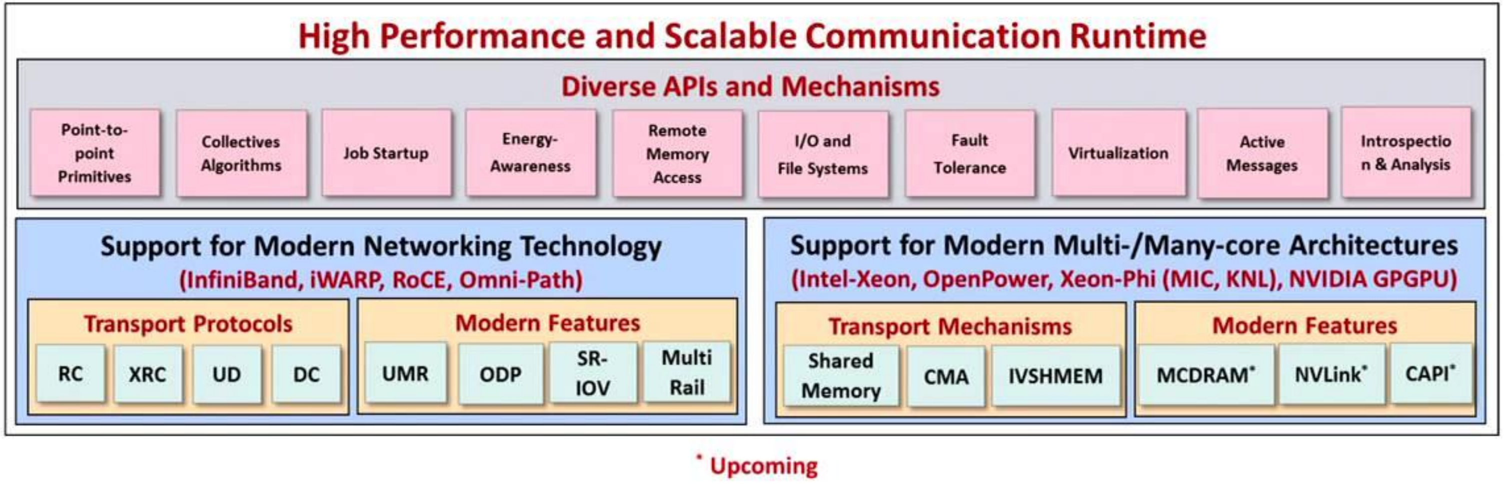
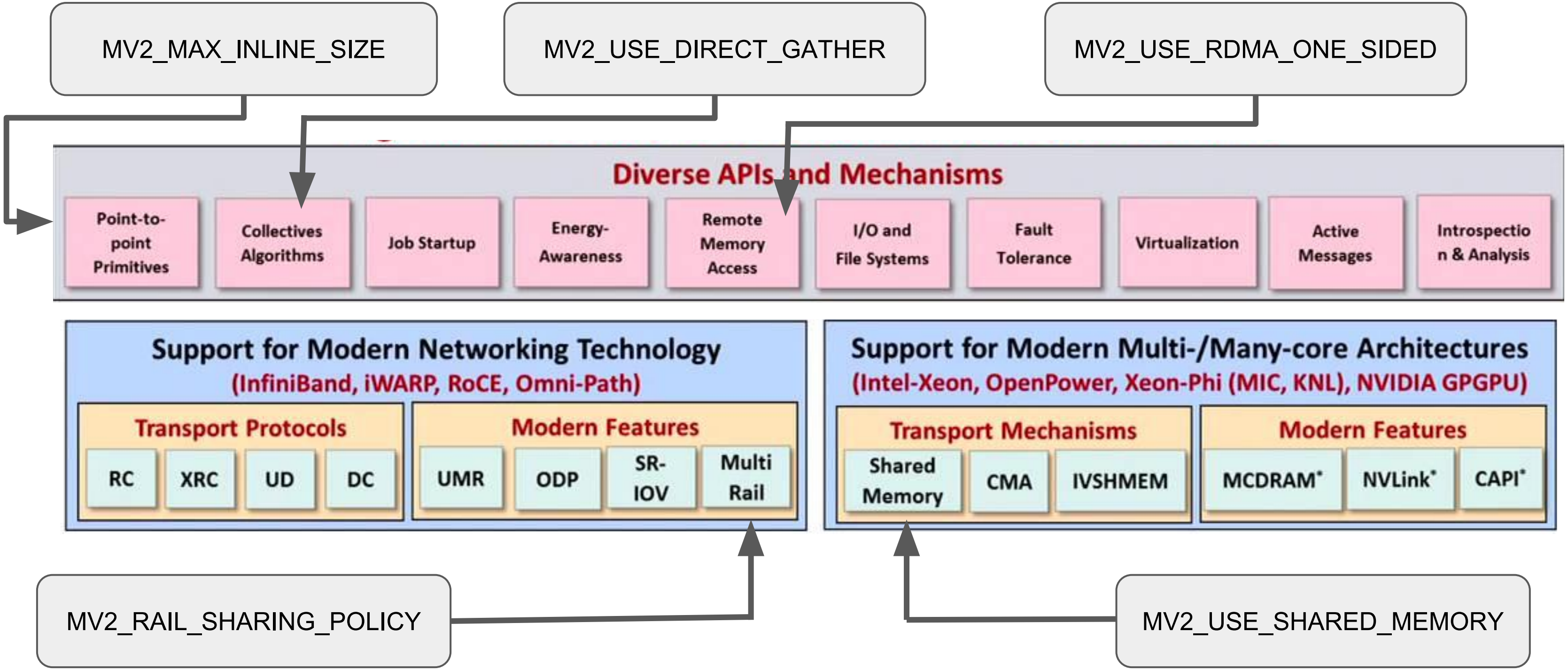



***Image credits: Dr. D.K. Panda, Network Based Computing Lab, The Ohio State University***

# MPI libraries are highly tunable

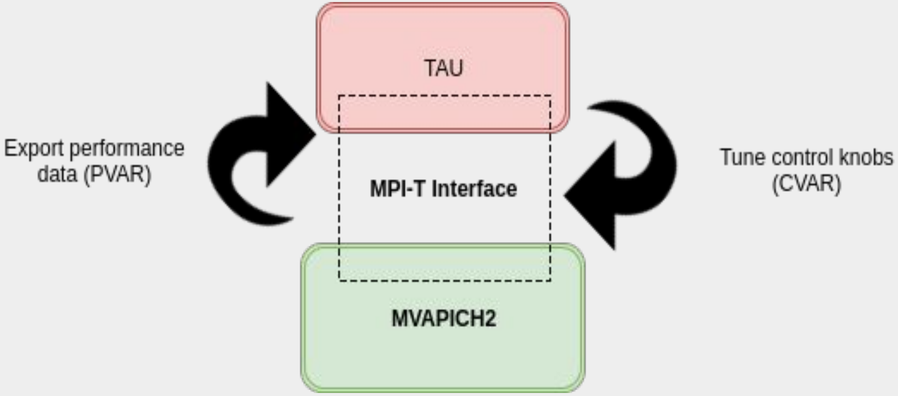
## MVAPICH2 offers ~160 tunable parameters across the stack



# MPI Tools Information Interface (MPI_T)

**Performance Variables (PVARs)**: Counters, MPI state info, etc

**Control Variables (CVARs)**: Knobs to re-configure MPI

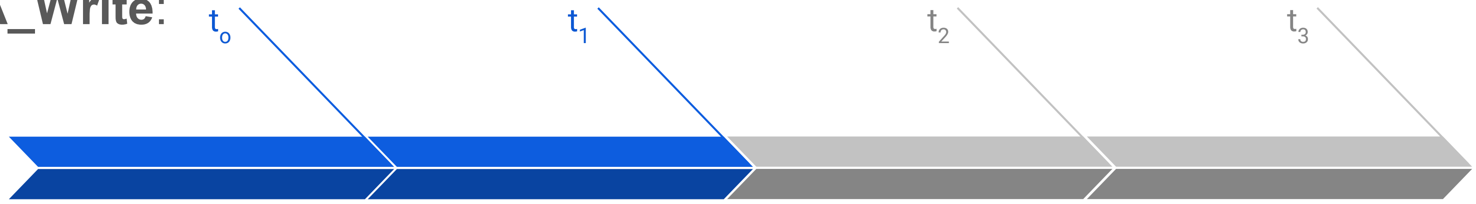


# MPI ***Rendezvous*** protocol over RDMA

- MVAPICH2 exports a CVAR (knob) to **statically** specify the MPI point-to-point rendezvous protocol over RDMA-enabled hardware (MV2_RNDV_PROTOCOL)
  - Rendezvous communication happens when data size > EAGER threshold
  - **RDMA Write**:
    - Sender sends out a “RTS (Request To Send)” to receiver, and “pushes” out data using RDMA Write once it receives ACK from receiver
  - **RDMA Read:**
    - Sender sends out an RTS along with its local data buffer information, and receiver “pulls” the data into its buffer using RDMA Read upon receipt of the RTS
- Turns out this knob can have a ***potentially*** significant impact on **non-blocking** point-to-point communication performance depending on the **runtime order** of the posting of MPI_Isend, MPI_Irecv and Wait operations

# Effect of event ordering on optimal protocol choice:

Consider this **global timeline** of a non-blocking point-to-point message using **RDMA_Write**:




**MPI_Irecv**

Receiver posts MPI_Irecv and carries on with “computation”

**MPI_Wait (receiver)**

Receiver posts MPI_Wait and blocks, waiting for RTS

**MPI_Isend**

Sender sends out RTS message and carries on with “computation”

**MPI_Wait (sender)**

Sender blocks waiting for ACK from receiver and eventually sends out data using RDMA Write

- By **default,** MVAPICH2 uses RDMA_Write for Rendezvous
- RDMA_Read for this simple benchmark for 2 processes over infiniband instead of RDMA_Write yields **12-16% better runtime for LARGE DATA transfers**! Why?

# 3DStencil: More Rendezvous protocol tuning results

- We consider a simple 3DStencil application that sends out “ghost” cells to 6 nearest neighbours using **non-blocking, point-to-point** messages of a **fixed size**
- In a loop, each process posts all non-blocking receives and sends first, then performs dummy computation for the period of time = pure communication time, and then performs an **MPI_Waitall()** for all “ghost” cell data to arrive.
- **RDMA_Read** improves runtime by **7-10%** as opposed to using **RDMA_Write (for all point-to-point calls)**
- Someone please ask me why RDMA_Read isn’t the default!

# Wait, what has this to do with TraceR?

# Motivation to use simulation

- Currently MVAPICH2 only supports a **static setting** for the RDMA-based Rendezvous protocol
  - No support to set this protocol differently for different processes - one static setting applied uniformly
  - No support to set this protocol at a per-message granularity
  - No support to tune it dynamically at runtime
  - No one even knows if tuning it at runtime would be useful!
- MPI_T interface supports message-level CVARs “in theory”, but no one actually uses it (yet!)

**Research goal:** Prove/Disprove that tuning the Rendezvous protocol at such a fine granularity **would indeed be useful**

# TraceR+CODES to the rescue!

- TraceR supports re-playing of OTF2 traces of real or synthetically generated MPI applications
- TraceR simulates certain portions of the MPI runtime of interest
  - Point-to-point: Both blocking and non-blocking routines are supported
  - Eager and Rendezvous protocol supported (more on that later)
  - Collectives are supported (although not relevant here)
  - MPI layer is modeled by adding “software delays” representing the latency of “going through” the MPI call-stack.
- CODES supports a variety of network topologies
  - Would be useful to test out the impact of runtime Rendezvous protocol tuning on a variety of hardware configurations

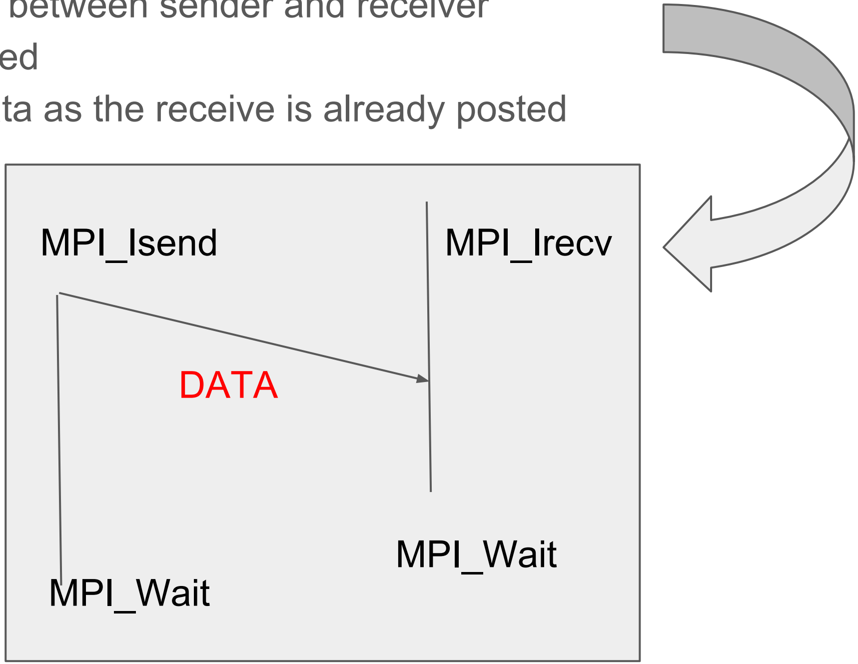
# TraceR: Existing Rendezvous protocol design

- Data is transferred "as soon as possible" (after receive and send are posted):
  - There is no explicit RTS-ACK message transfer between sender and receiver
  - Receiver notifies the sender that receive is posted
  - If the sender is late, it immediately sends the data as the receive is already posted

**Communication Progress**:

TraceR implicitly assumes that there is a "background" thread/mechanism that responds to communication requests from other PE's (LP's)

Thus, PE's don't necessarily depend on MPI_Wait's to process pending communication requests

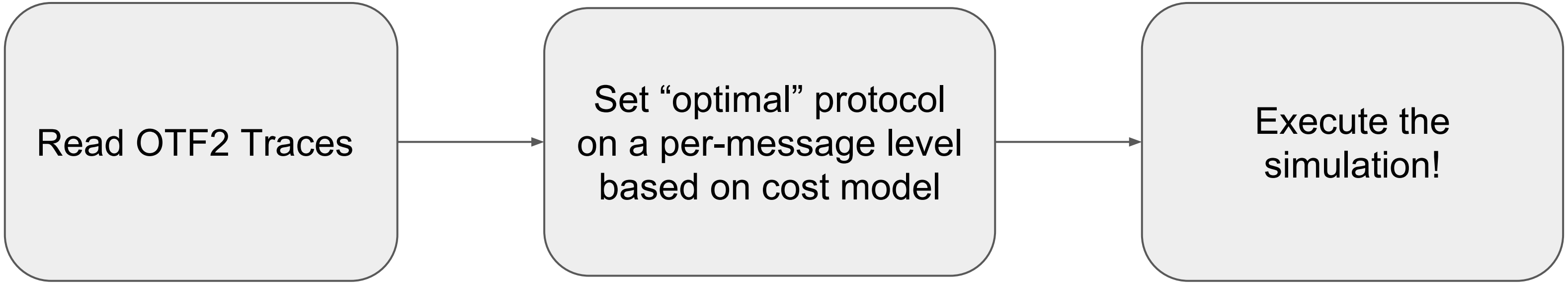


# TraceR: Implementing RDMA_Write

- Firstly, we need to remove the notion of background "threads":
  - Real MPI implementations do NOT support threading for communication progress as default - it is proven to be expensive in many scenarios
- Create explicit "RTS" and "ACK" message types between PE's in TraceR
  - Store these messages on either end until a corresponding MPI_Isend/Irecv or Wait is posted
- Check for the receipt of "RTS" and "ACK" messages inside MPI_Wait and respond accordingly
- Currently being tested:
  - Application performance expected to be slower than with TraceR's default implementation as overall communication latency is bound to increase
  - Only implementing the "conservative" PDES simulation for now as we don't necessarily care too much about TraceR performance for the time being
  - RDMA_Read design is still being worked over

# Putting it all together

- The research goal is to replay "what-if" scenarios for when the "ideal" RDMA Rendezvous protocol is implemented on a per-message level
- Exactly why TraceR could be useful - we have a control over the setting of the RDMA protocol on a per-message level
  - Need to develop a simple cost model for the data transfer times under the two protocols, so that we can decide what the optimal protocol in a given situation is!



# Looking even further...

- Our initial goal is to use TraceR as an “oracle” that knows the optimal protocol on a per message-level
- Eventually, we’d like to implement this tuning logic as the simulation is running
- Profile the simulation as it is running, and choose the optimal protocol based on collected timing data
- A more accurate scenario given that we eventually want to use a tool to do the tuning of an actual MPI application

Thanks! Questions?

sramesh@cs.uoregon.edu