



Enabling Exascale Co-Design Architecture

Devendar Bureddy

August 2016

Mellanox Connects the World's Fastest Supercomputer



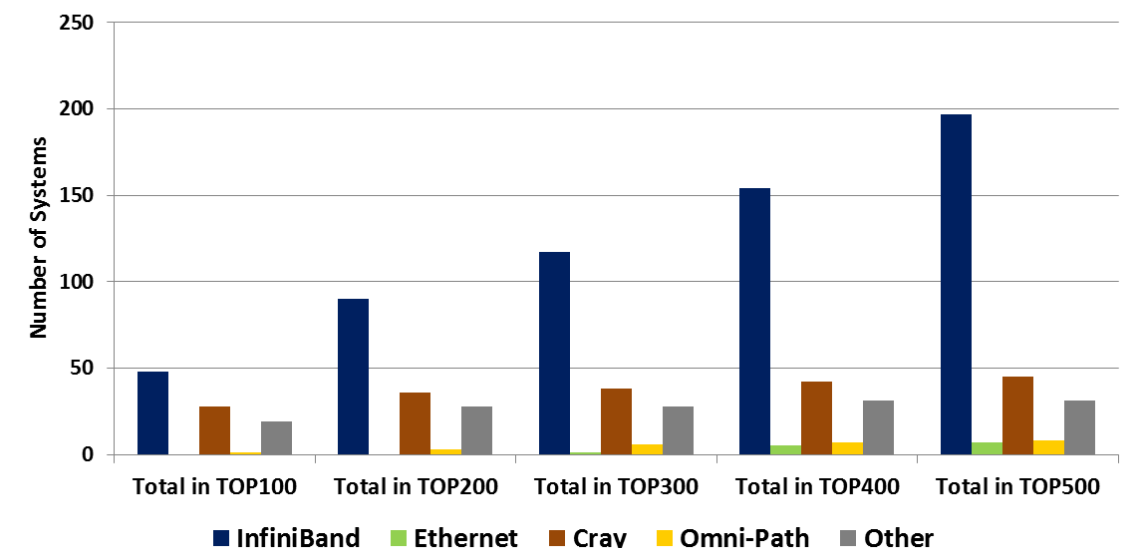
#1 on the TOP500 Supercomputing List

- 93 Petaflop performance, 3X higher versus #2 on the TOP500
- 40K nodes, 10 million cores, 256 cores per CPU
- Mellanox adapter and switch solutions

- The TOP500 list has evolved, includes HPC & Cloud / Web2.0 Hyperscale systems
- Mellanox connects 41.2% of overall TOP500 systems
- Mellanox connects 70.4% of the TOP500 HPC platforms
- Mellanox connects 46 Petascale systems, Nearly 50% of the total Petascale systems

**InfiniBand is the Interconnect of Choice for
HPC Compute and Storage Infrastructures**

TOP500 - TOP 100, 200, 300, 400, 500 Systems Distribution
HPC Systems Only



The Ever Growing Demand for Higher Performance

Performance Development

Terascale



Petascale

1st



"Roadrunner"



Exascale

OAK RIDGE
National Laboratory
"Summit" System

Lawrence Livermore
National Laboratory
"Sierra" System

2000

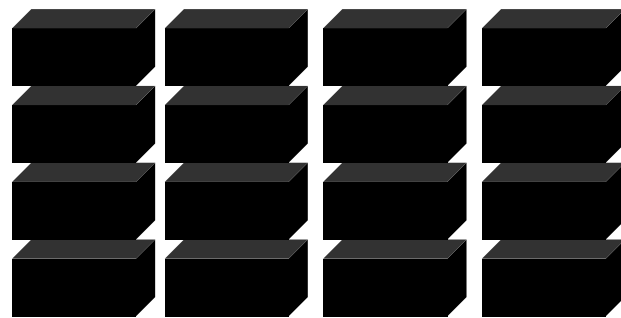
2005

2010

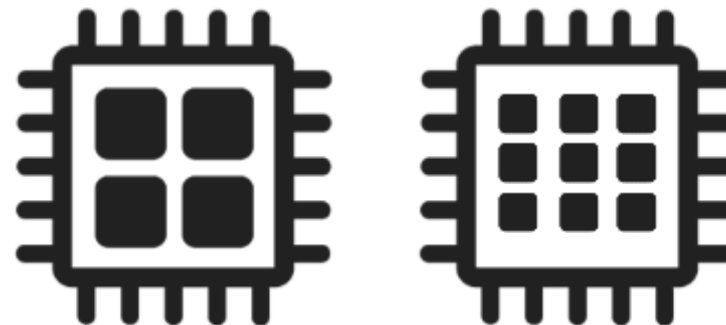
2015

2020

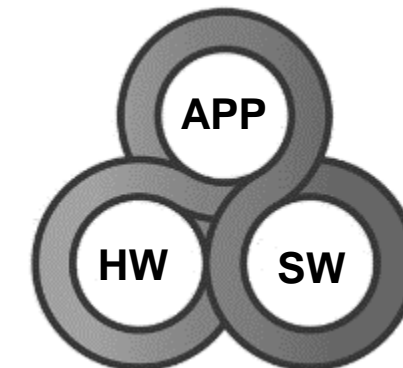
The Interconnect is the Enabling Technology



SMP to Clusters



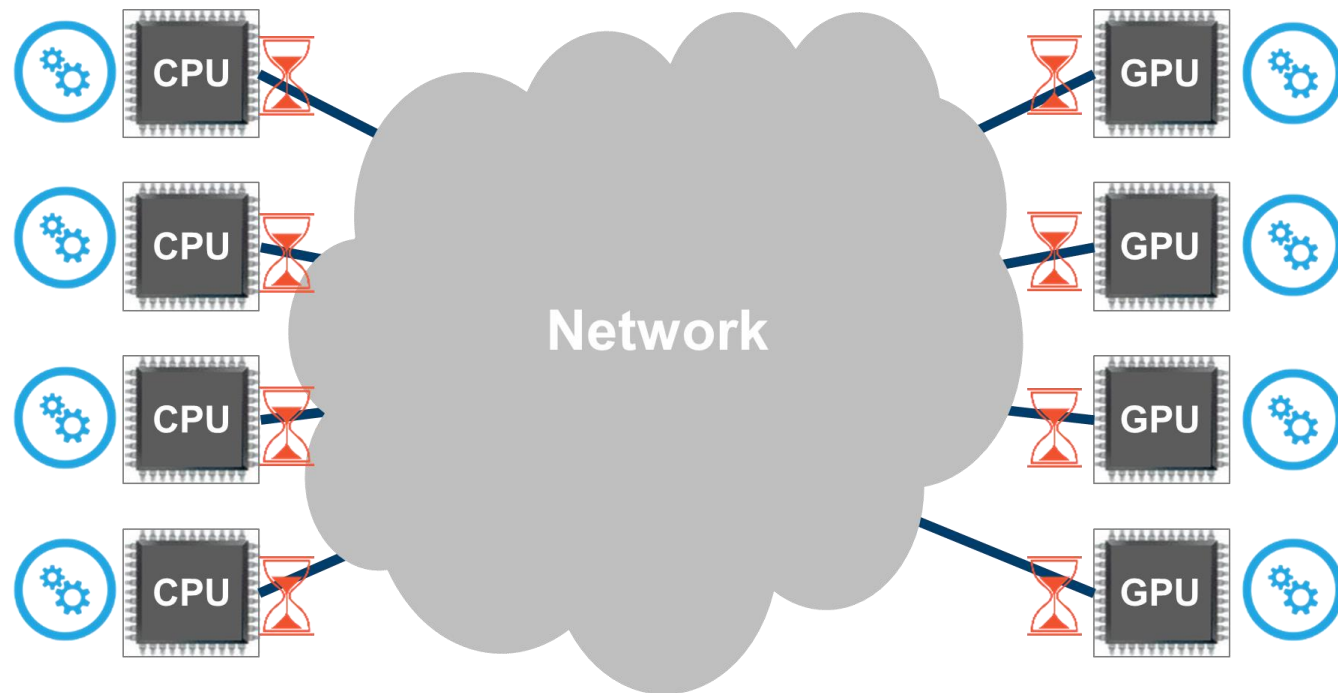
Single-Core to Many-Core



Application
Software
Hardware

Co-Design

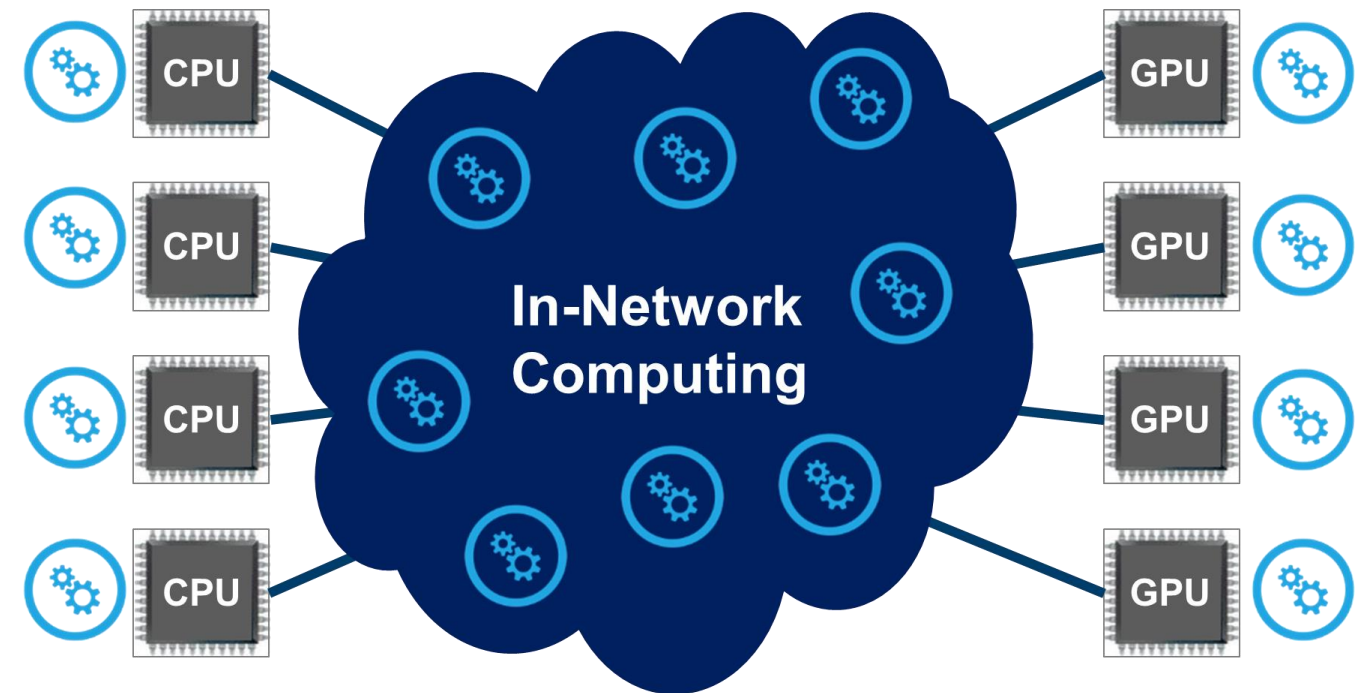
CPU-Centric



Limited to Main CPU Usage
Results in Performance Limitation

**Must Wait for the Data
Creates Performance Bottlenecks**

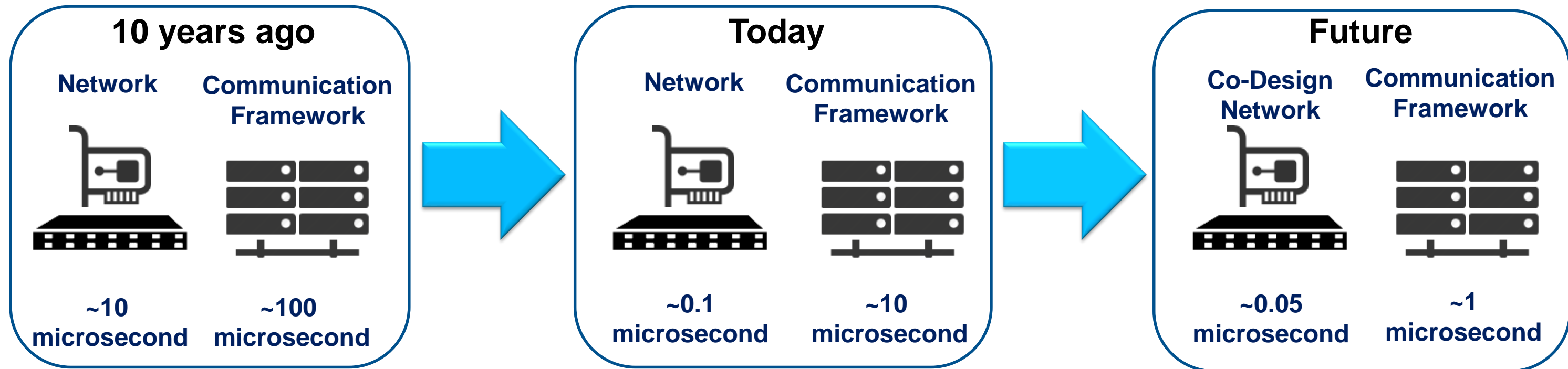
Co-Design



Creating Synergies
Enables Higher Performance and Scale

**Work on The Data as it Moves
Enables Performance and Scale**

Breaking the Application Latency Wall



- Today: Network device latencies are on the order of 100 nanoseconds
- Challenge: Enabling the next order of magnitude improvement in application performance
- Solution: Creating synergies between software and hardware – intelligent interconnect

Intelligent Interconnect Paves the Road to Exascale Performance

Highest-Performance 100Gb/s Interconnect Solutions

Adapters

ConnectX[®] 5

100Gb/s Adapter, 0.6us latency
200 million messages per second
(10 / 25 / 40 / 50 / 56 / 100Gb/s)



Switch

SwitchIB[™] 2

36 EDR (100Gb/s) Ports, <90ns Latency
Throughput of 7.2Tb/s
7.02 Billion msg/sec (195M msg/sec/port)



Switch

Spectrum[™]

32 100GbE Ports, 64 25/50GbE Ports
(10 / 25 / 40 / 50 / 100GbE)
Throughput of 6.4Tb/s



Interconnect

LinkX[™]

Transceivers
Active Optical and Copper Cables
(10 / 25 / 40 / 50 / 56 / 100Gb/s)



VCSELs, Silicon Photonics and Copper

Software

HPC-X[™]

MPI, SHMEM/PGAS, UPC
For Commercial and Open Source Applications
Leverages Hardware Accelerations

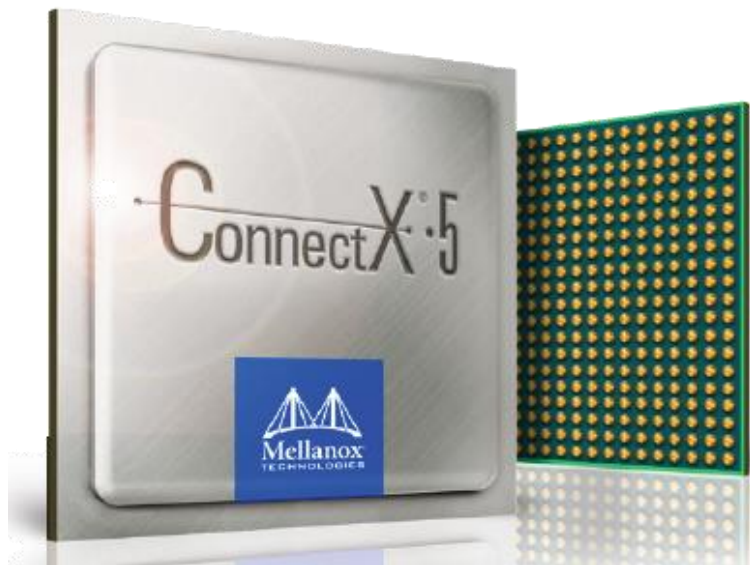


ConnectX-5 EDR 100G Advantages



NEW!

ConnectX[®]·5



Performance

100Gb/s Throughput
0.6usec Latency (end-to-end)
200M Messages per Second

Smart

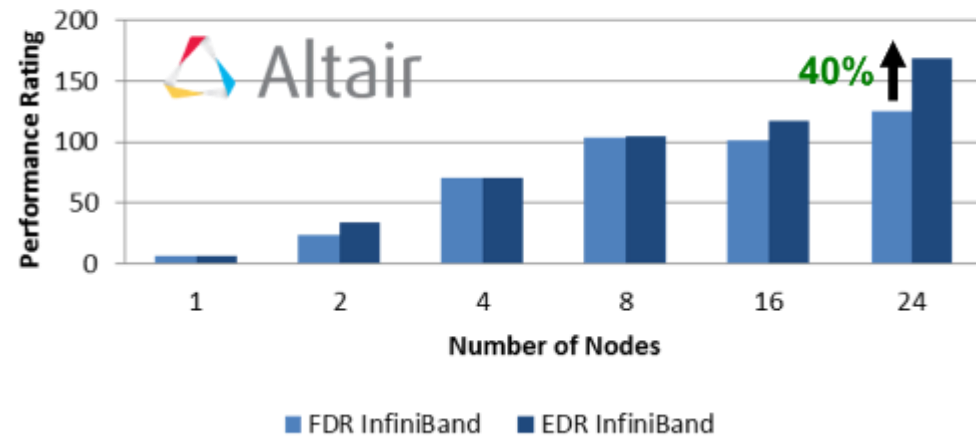
MPI Collectives in Hardware
MPI Tag Matching in Hardware
In-Network Memory

Platform

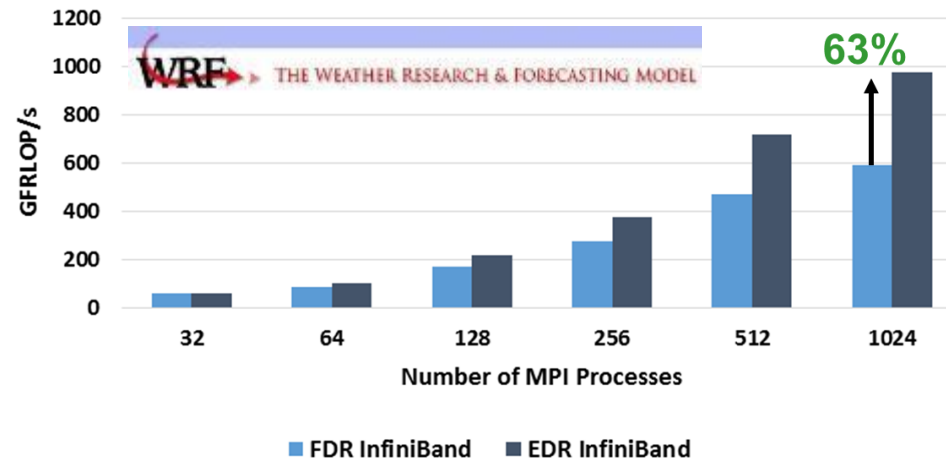
PCIe Gen3 and Gen4
Integrated PCIe Switch
Advanced Dynamic Routing

The Performance Advantage of EDR 100G InfiniBand (28-80%)

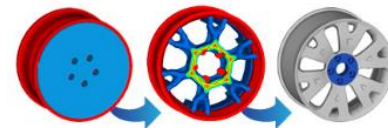
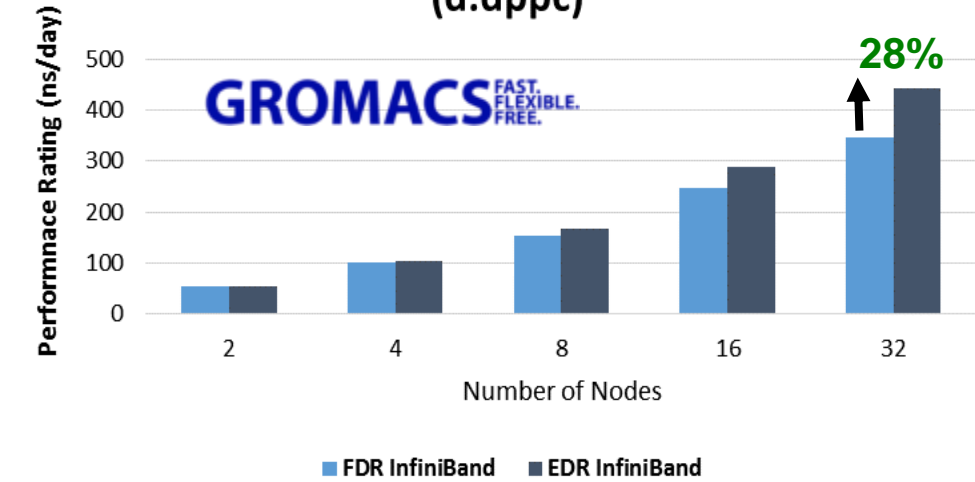
OptiStruct Performance (Engine_Assy.fem)



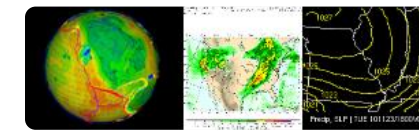
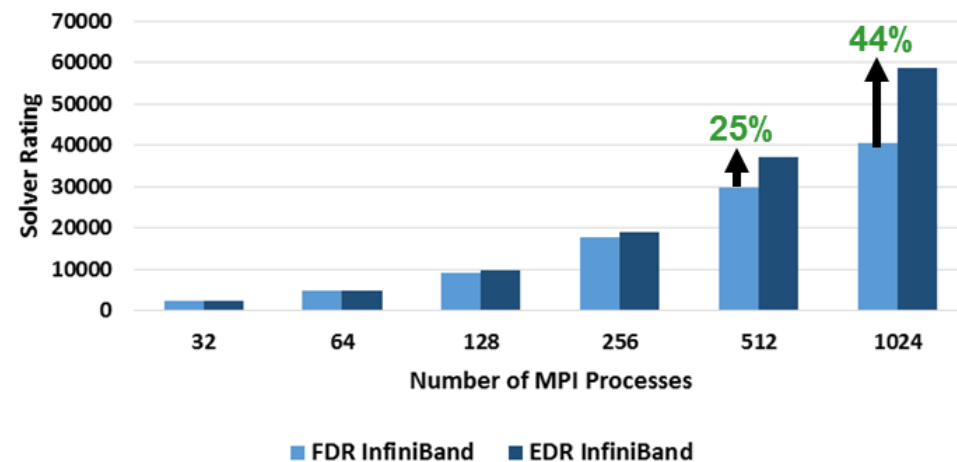
WRF Performance (conus12km)



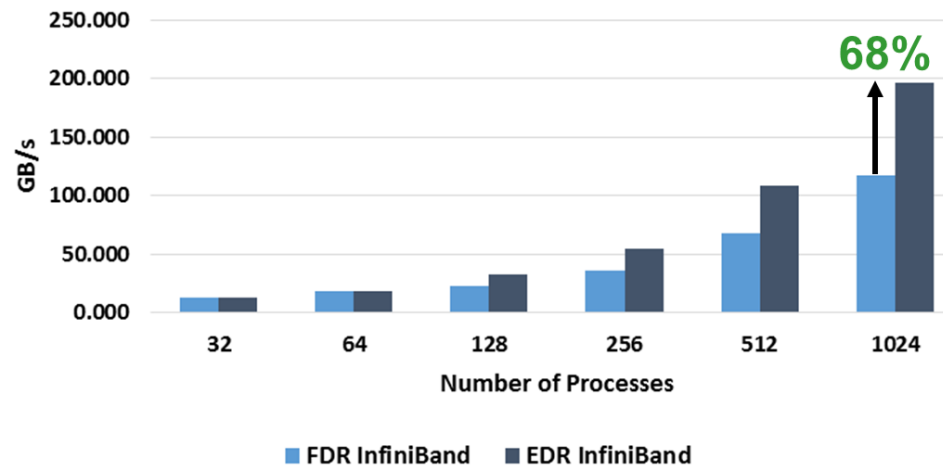
GROMACS Performance (d.dppc)



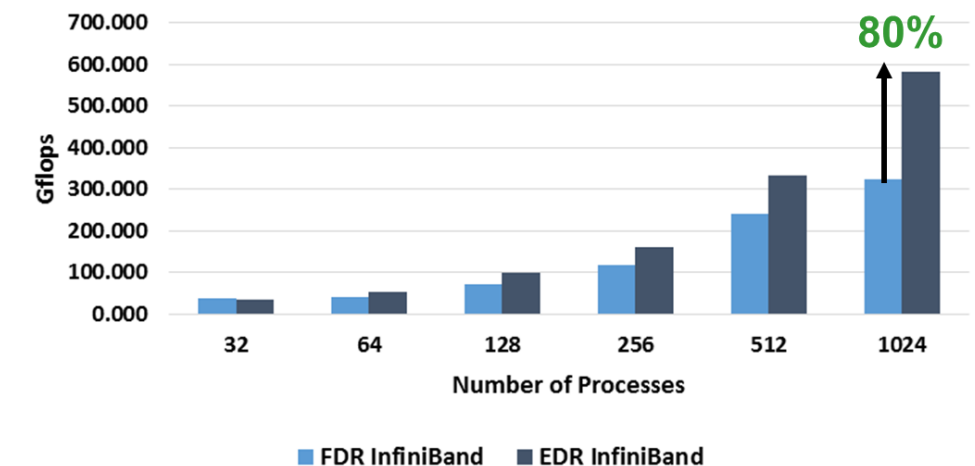
ANSYS Fluent 16.0 Performance (sedan_4m)



HPCC Performance (PTRANS_GB/s)



HPCC Performance (MPIFFT)



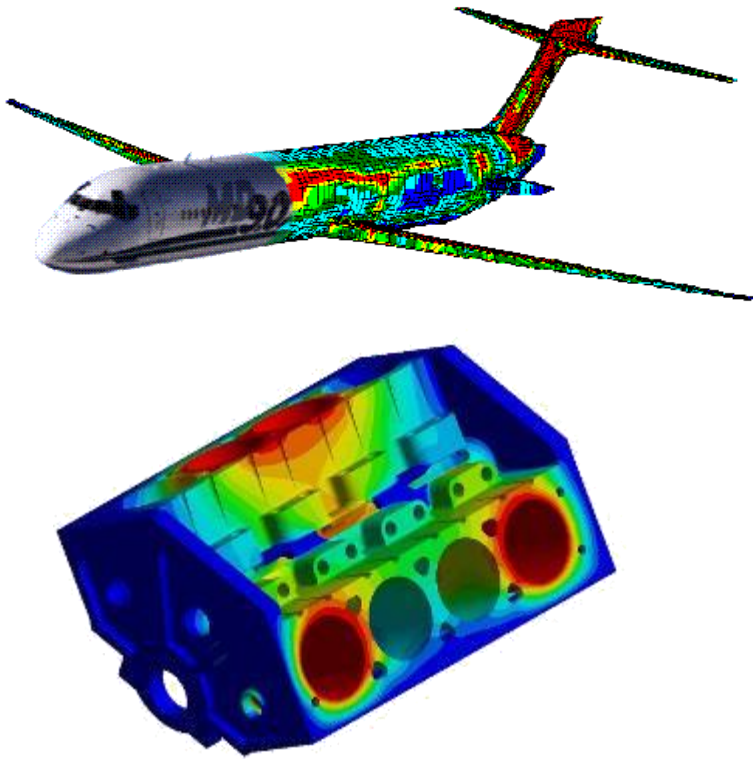
SHArP Enables Switch-IB 2 to Manage and Execute MPI Operations in the Network

Switch-IB 2 Enables the Switch Network to Operate as a Co-Processor

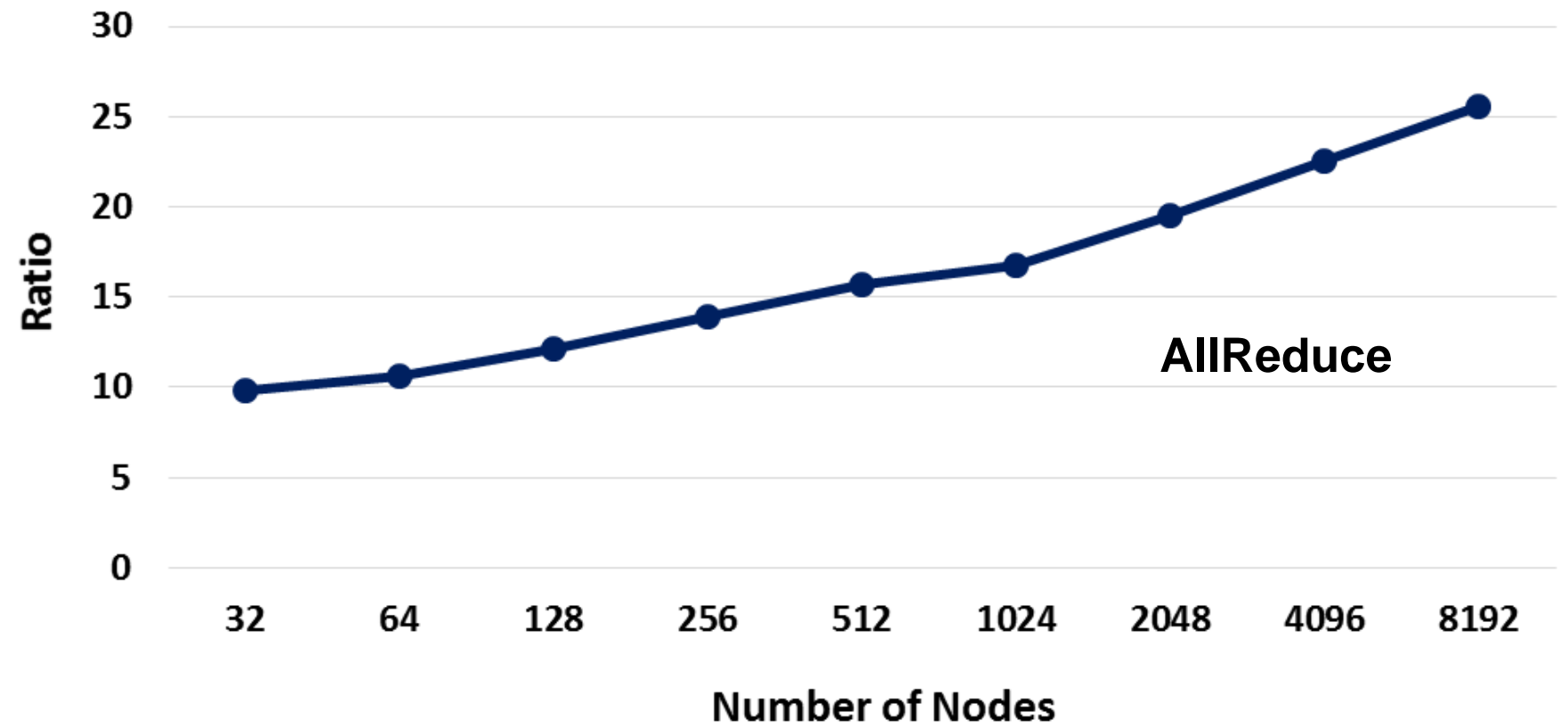
- The world fastest switch with <90 nanosecond latency
- 36-ports, 100Gb/s per port, 7.2Tb/s throughput, 7.02 Billion messages/sec
- Adaptive routing, congestion control
- Multiple topologies



- MiniFE is a Finite Element mini-application
 - Implements kernels that represent implicit finite-element applications



CPU-based versus Switch Collectives Offloads MiniFE Application - Latency Ratio (8 Bytes)



SHArP Performance Data – OSU Allreduce 1PPN, 128 nodes



Message Size [B]	SHArP based	Host Based	SHArP improvement factor
8	2.76	5.82	2.11
16	2.76	5.91	2.14
32	2.86	6.04	2.11
64	3.01	6.76	2.25
128	3.24	7.37	2.27
256	3.50	8.99	2.57
512	4.06	11.11	2.74
1024	5.49	18.04	3.29
2048	8.44	33.61	3.98
4096	14.48	46.93	3.24

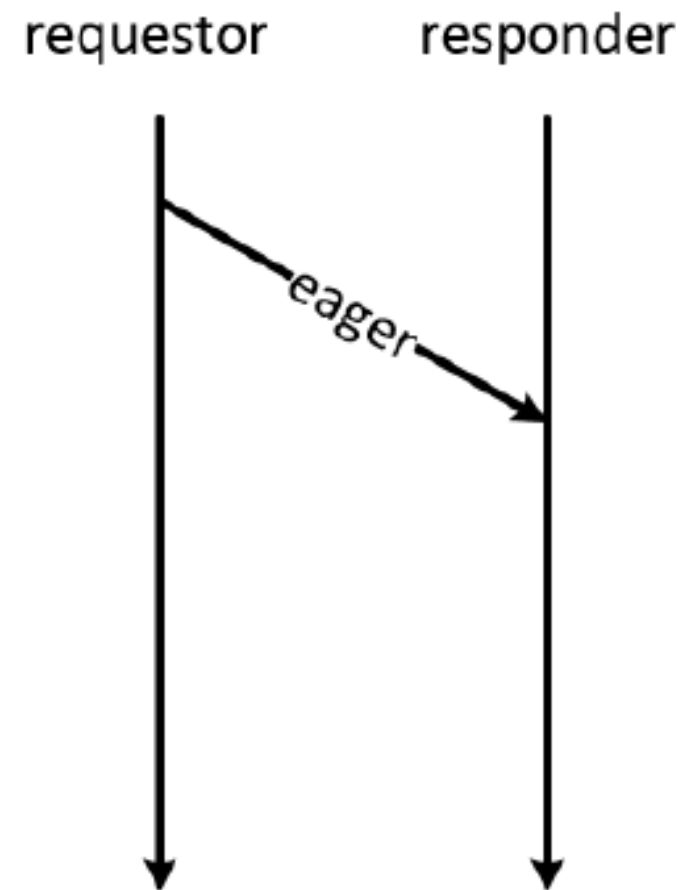
ConnectX-5: MPI Tag-Matching Support

- Sender: tag, communicator, destination, source (implicit)
- Receiver: tag (may be wild carded), communicator, source (may be wild carded), destination (implicit)
- Matching: sender and receiver envelopes must match
- Matching Ordering:
 - Matching envelopes are required
 - Posted received must be matched in-order against the in-order posted sends

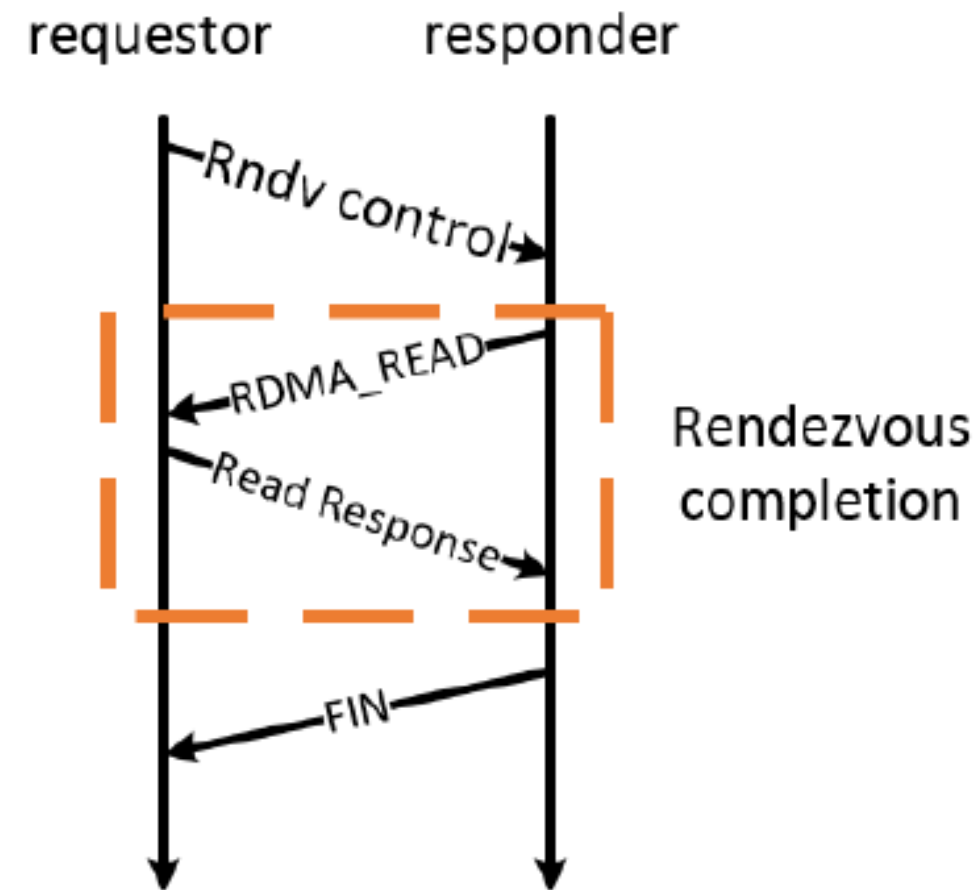


- Offloaded to the ConnectX-5 HCA
 - Full MPI tag-matching : tag matching as compute is progressing
 - Rendezvous offload : large data delivery as compute is progressing
- Control can be passed between Hardware and Software
- Verbs Tag-Matching support being up-streamed
- Implementation: Work in progress in UCX

Eager



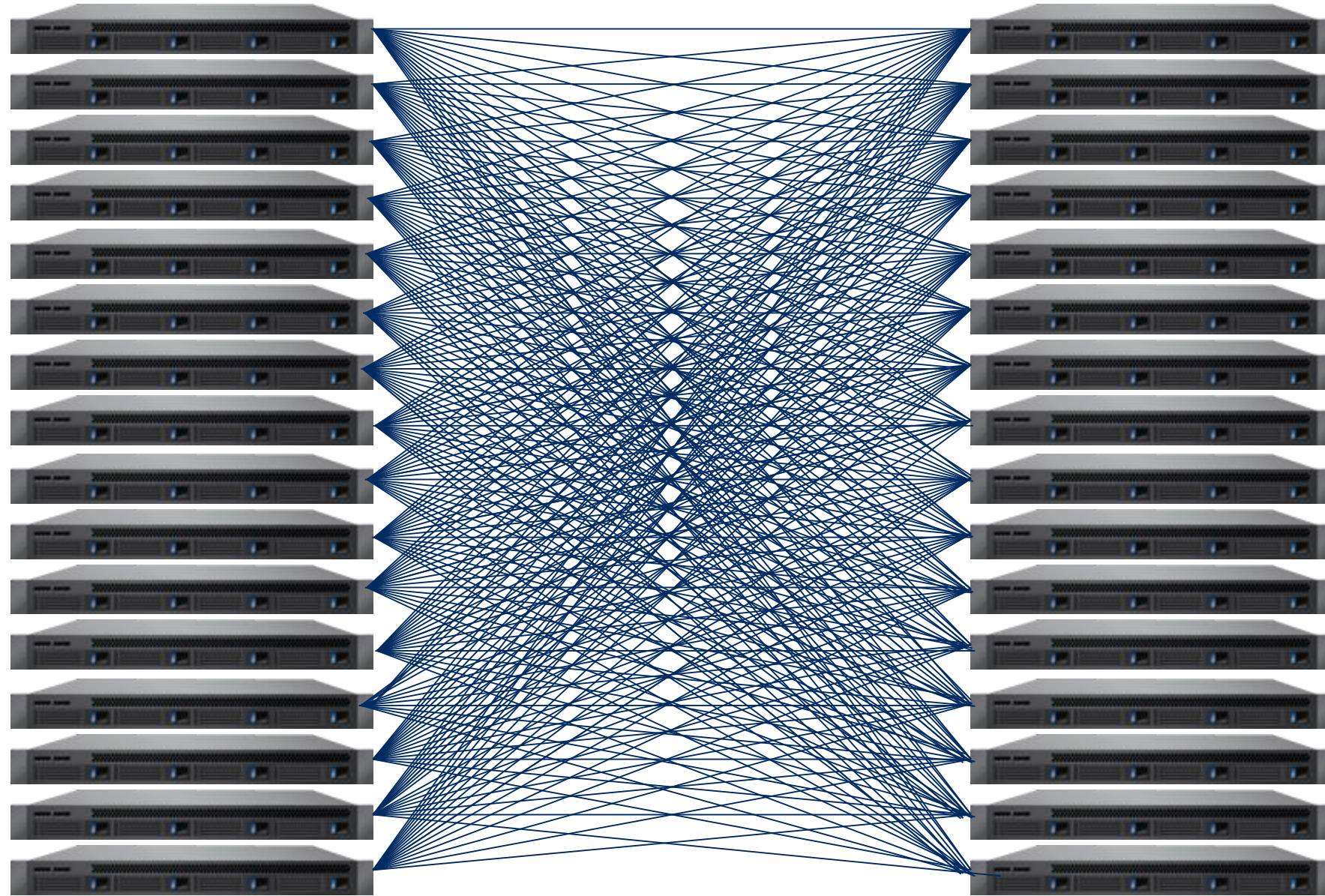
rendezvous



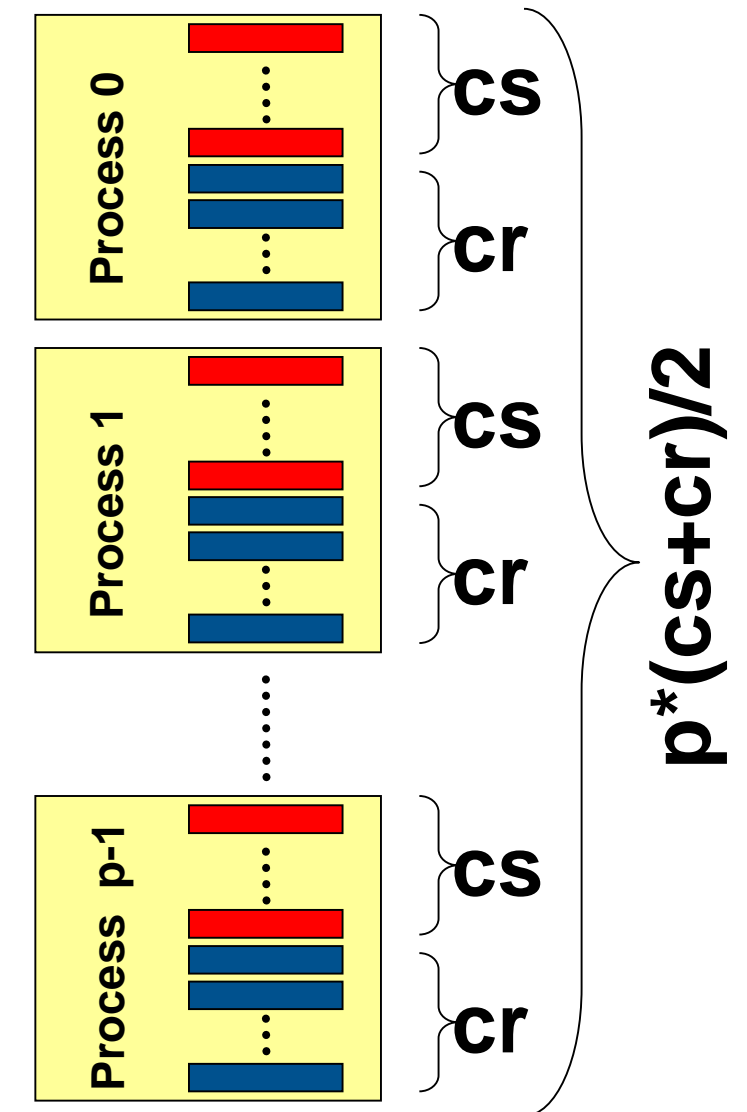
Dynamically Connected Transport

A Scalable Transport

Reliable Connection Transport Mode

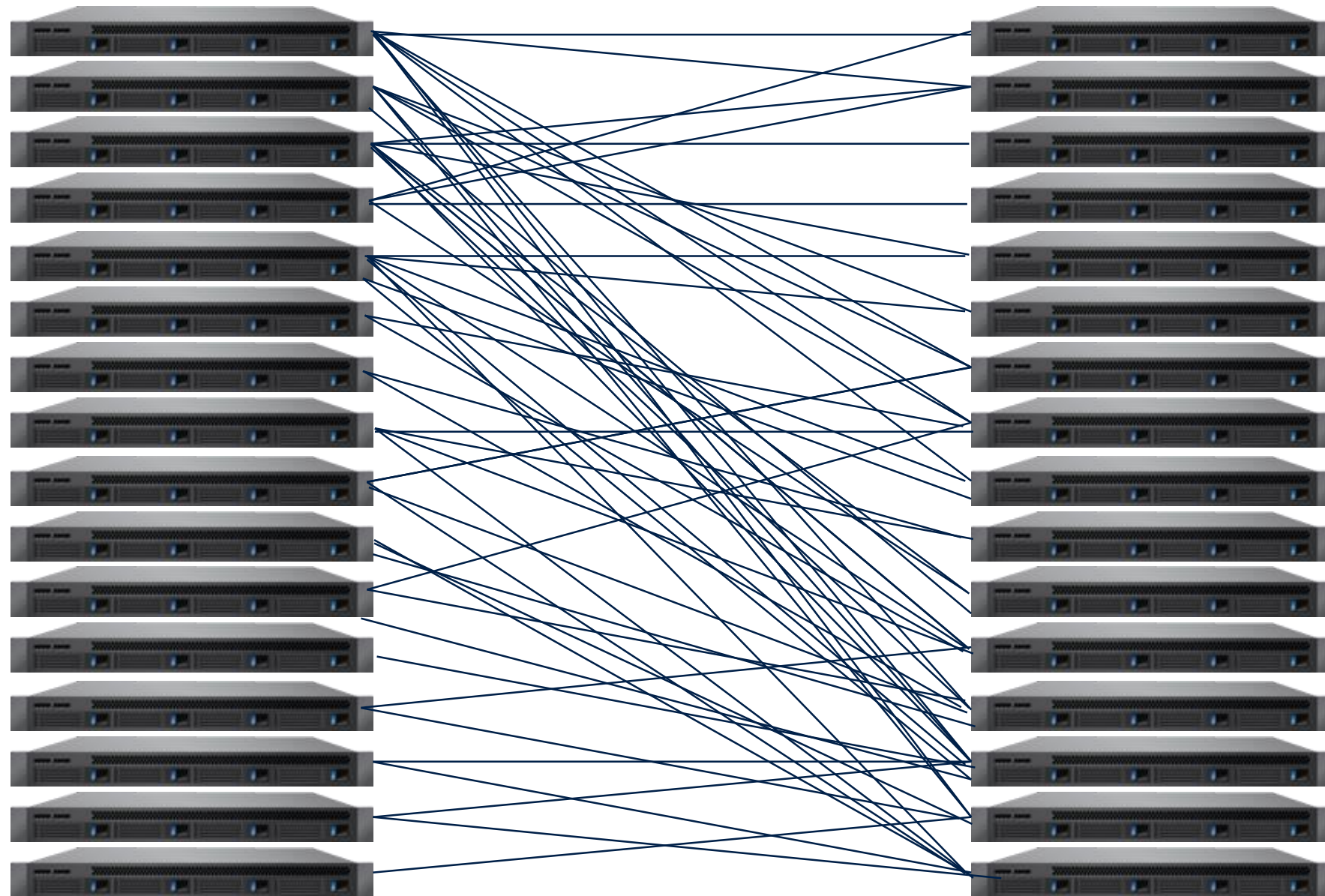


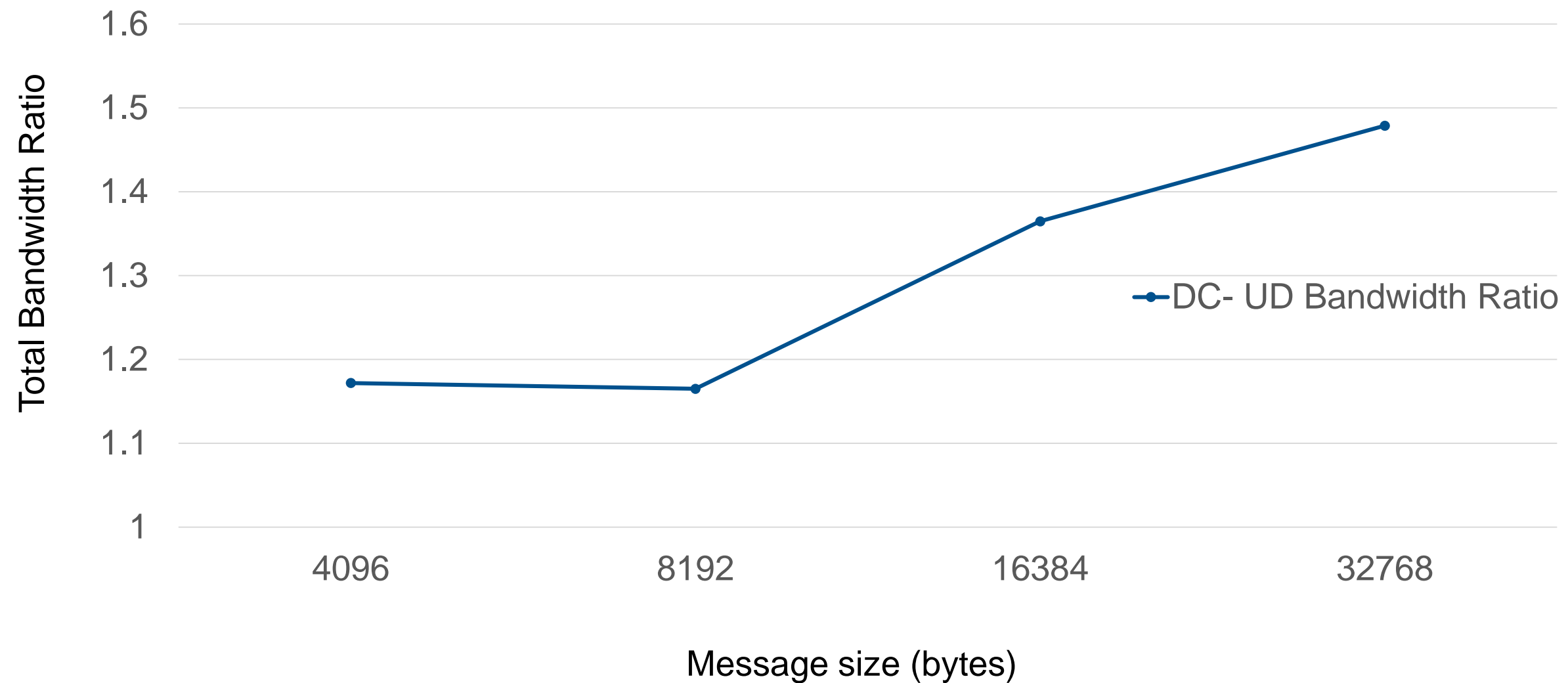
- Dynamic Connectivity
- Each DC Initiator can be used to reach any remote DC Target
- No resources' sharing between processes
 - process controls how many (and can adapt to load)
 - process controls usage model (e.g. SQ allocation policy)
 - no inter-process dependencies
- Resource footprint
 - Function of HCA capability
 - Independent of system size
- Fast Communication Setup Time



cs – concurrency of the sender
cr=concurrency of the responder

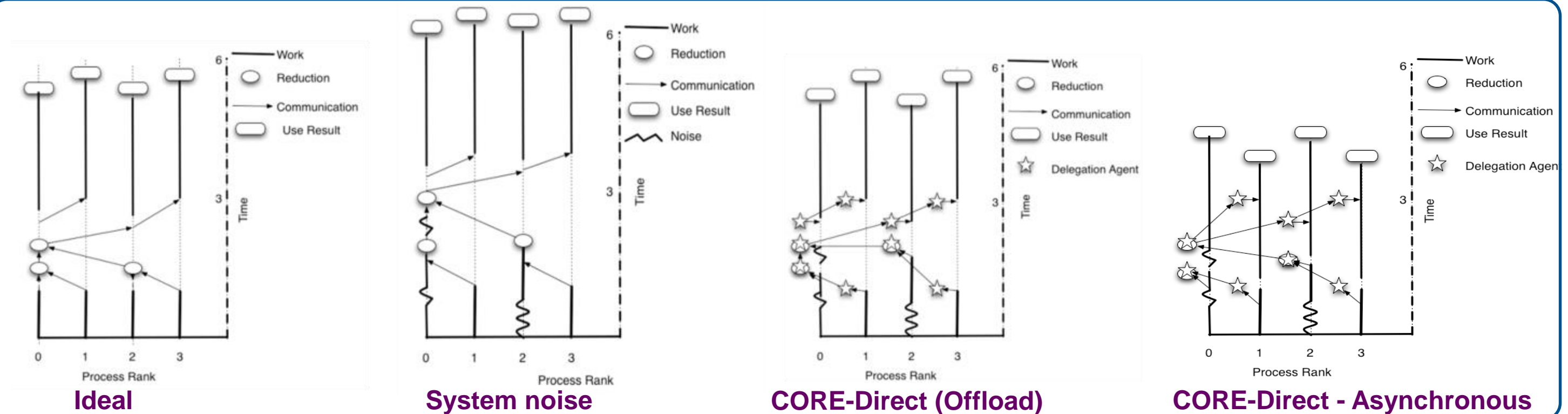
Dynamically Connected Transport Mode





Cross Channel Synchronization(aka CORE-Direct)

Smart Offloads for MPI/SHMEM/PGAS/UPC Collective Operations



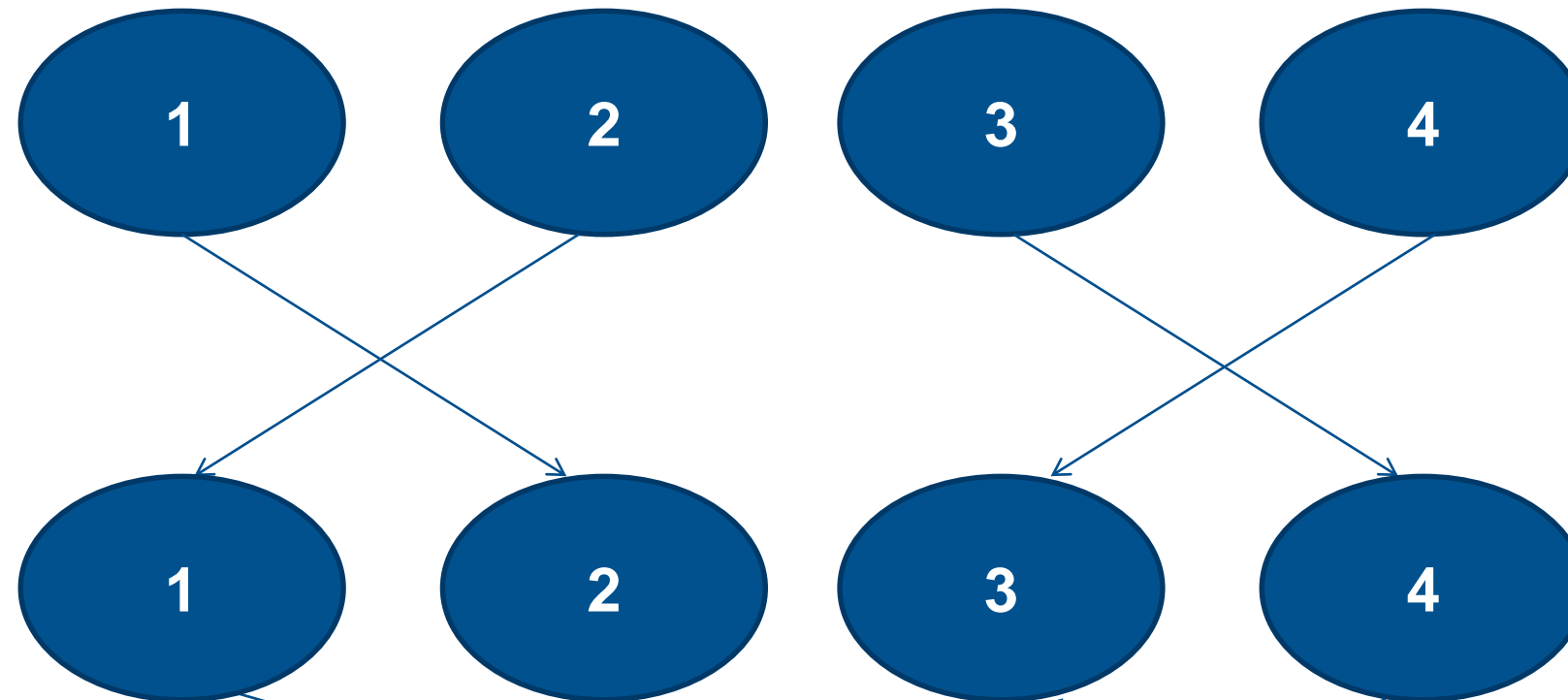
- CORE-Direct Technology
- Adapter-based hardware offloading for collective operations
- Includes floating-point capabilities on the adapters

- Task list
- Target QP for task
- Operation
 - Send
 - Wait for completions
 - Enable
 - Calculate

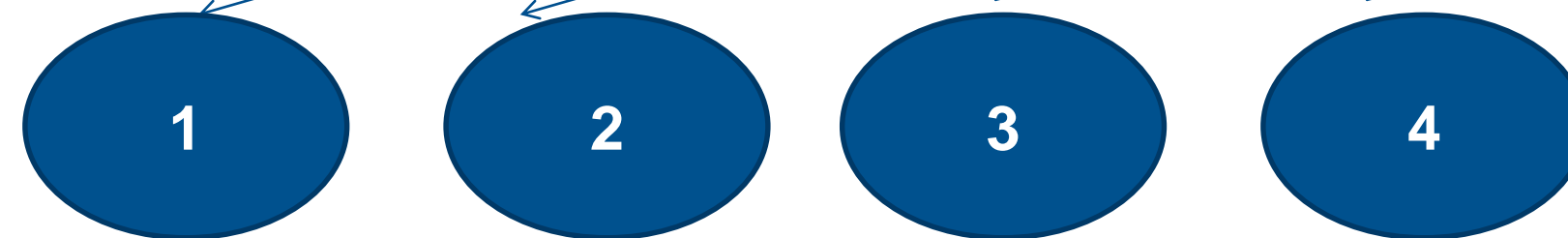


Example – Four Process Recursive Doubling

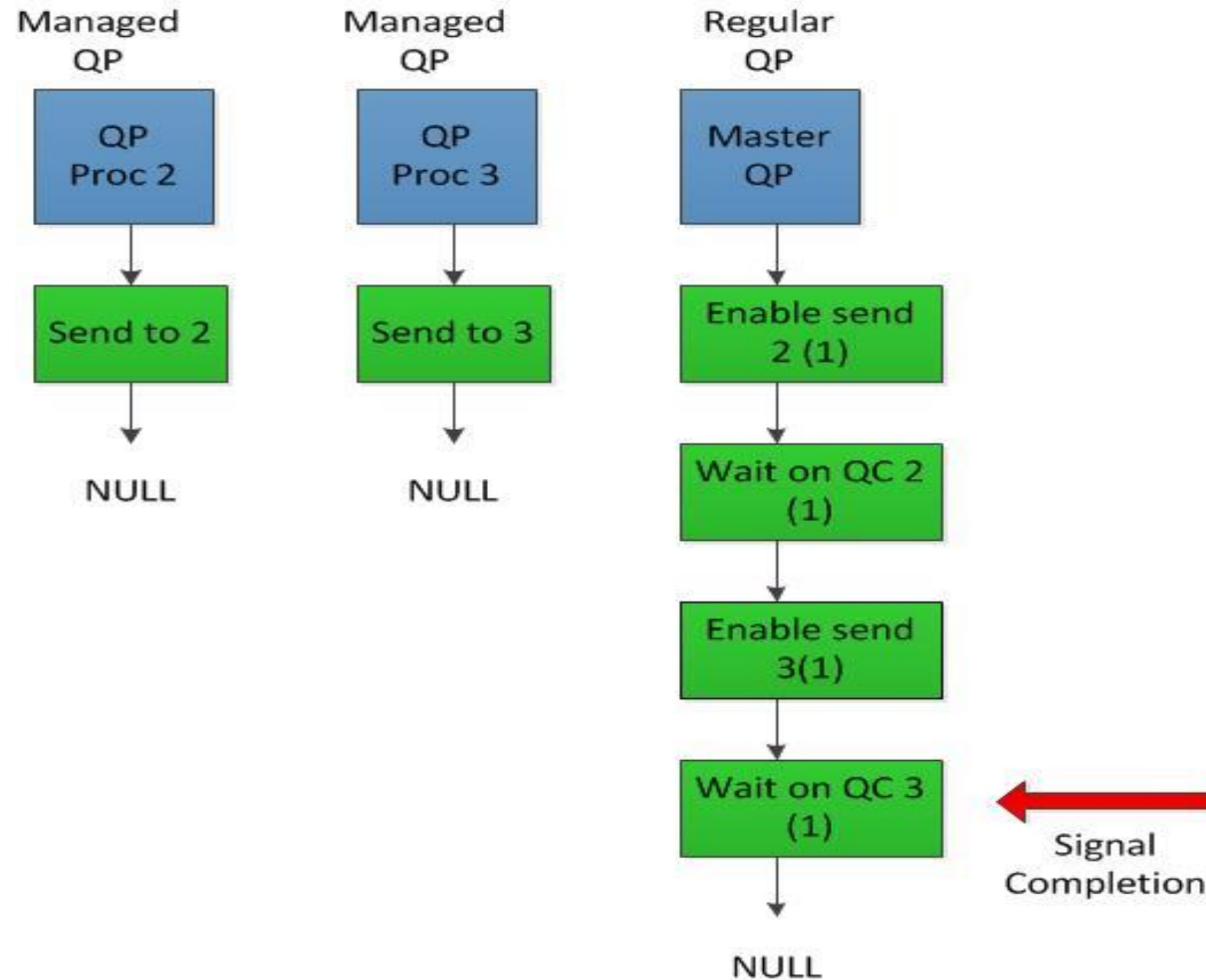
Step 1



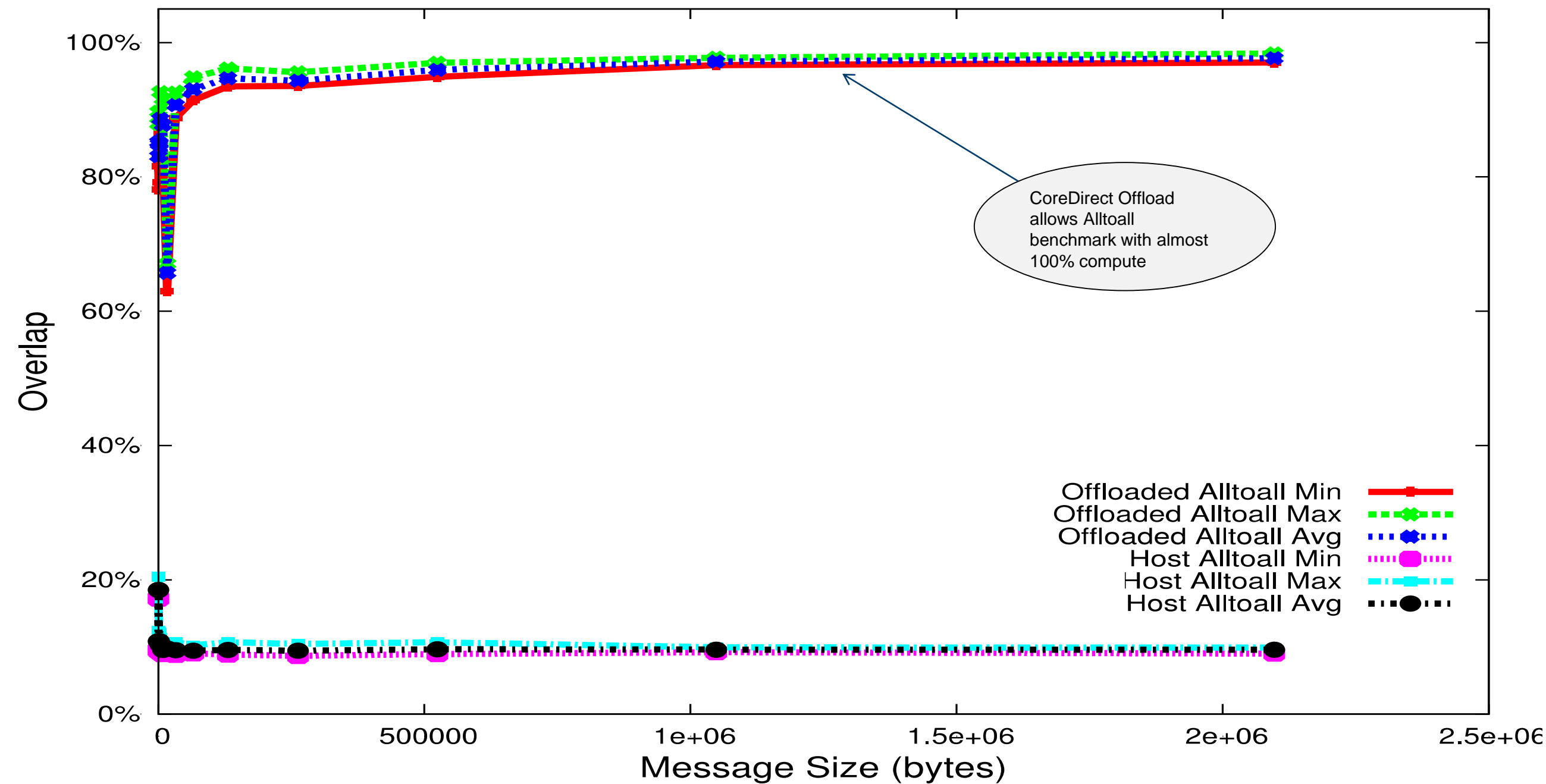
Step 2



Four Process Barrier Example – Using Managed Queues – Rank 0



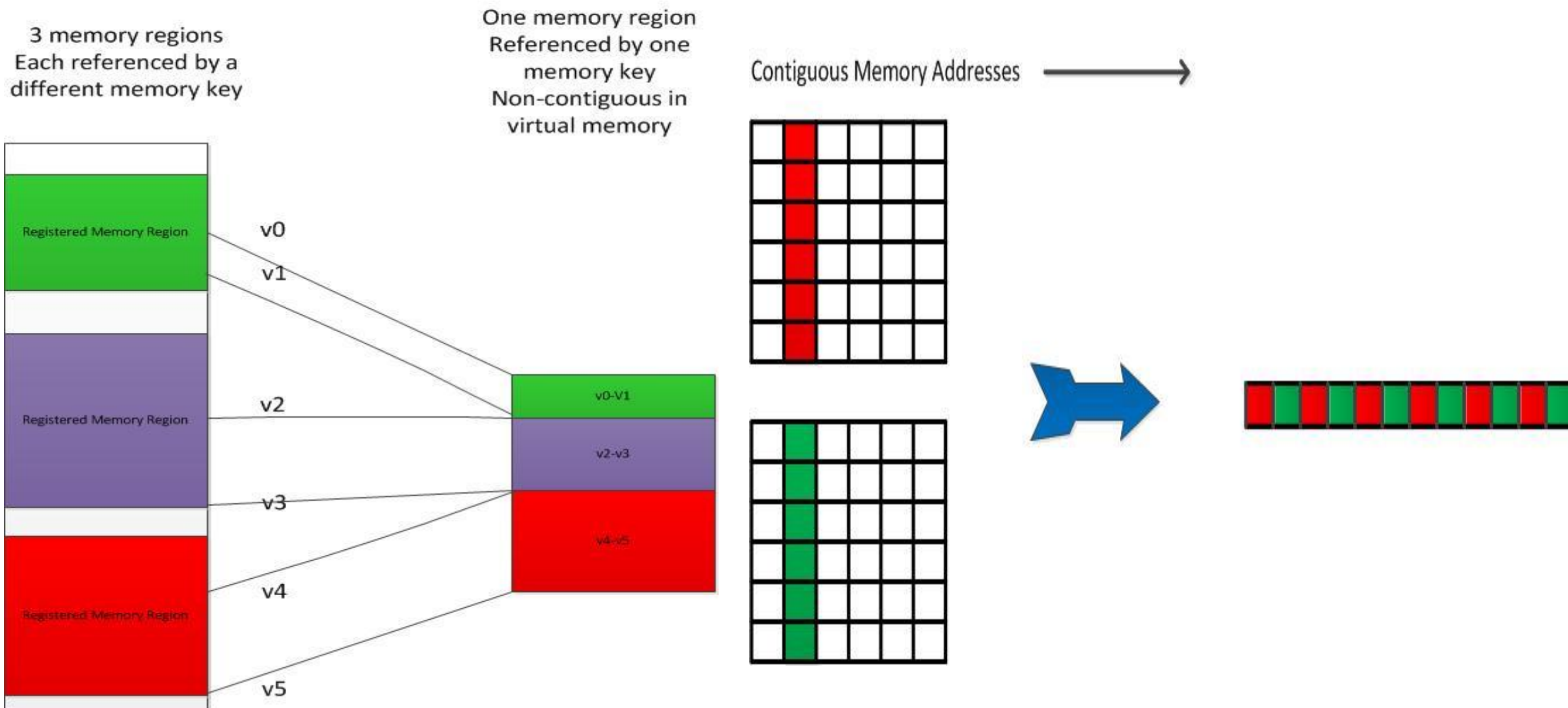
Nonblocking Alltoall (Overlap-Wait) Benchmark



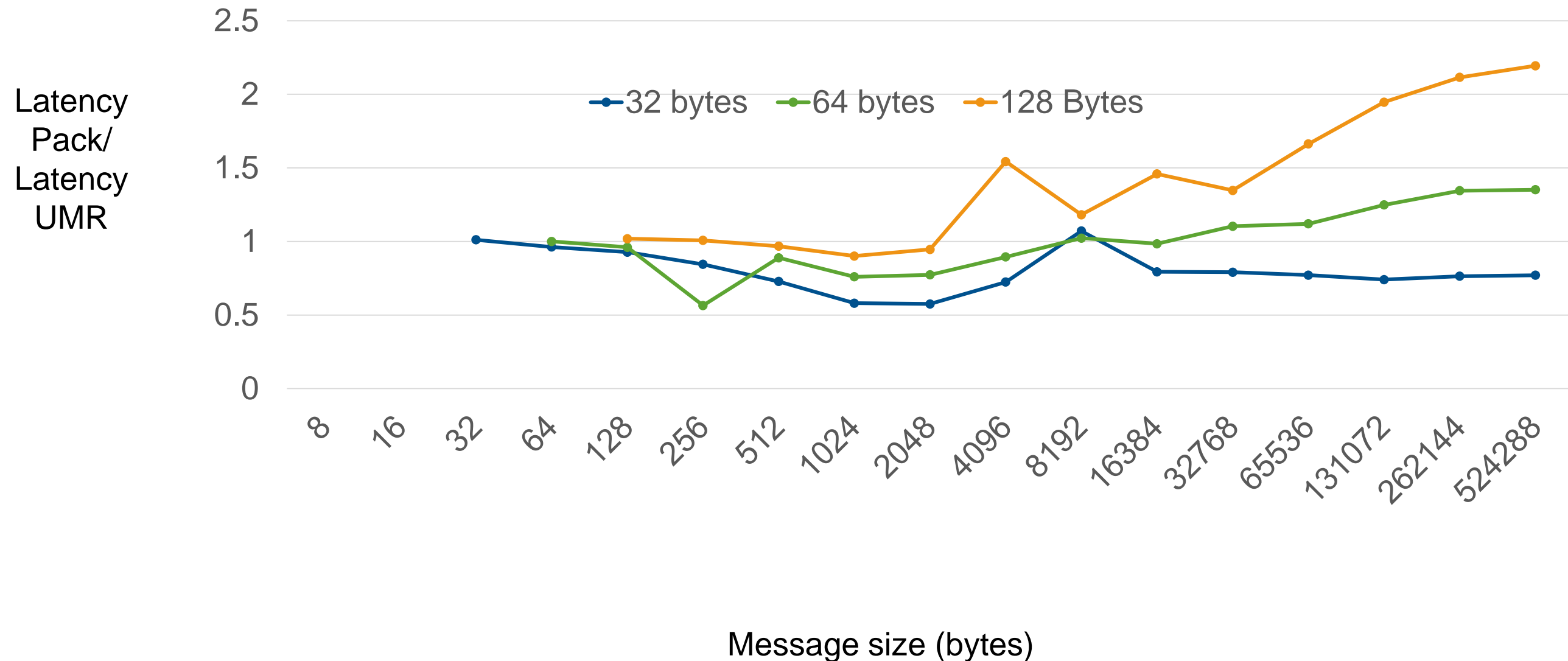
Non-Contiguous Data

- Support combining contiguous registered memory regions into a single memory region. H/W treats them as a single contiguous region (and handles the non-contiguous regions)
- For a given memory region, supports non-contiguous access to memory, using a regular structure representation – base pointer, element length, stride, repeat count.
 - Can combine these from multiple different memory keys
- Memory descriptors are created by posting WQE's to fill in the memory key
- Supports local and remote non-contiguous memory access
 - Eliminates the need for some memory copies

Optimizing Non Contiguous Memory Transfers



Hardware Gather/Scatter Capabilities – Regular Structure – Ping-Pong latency

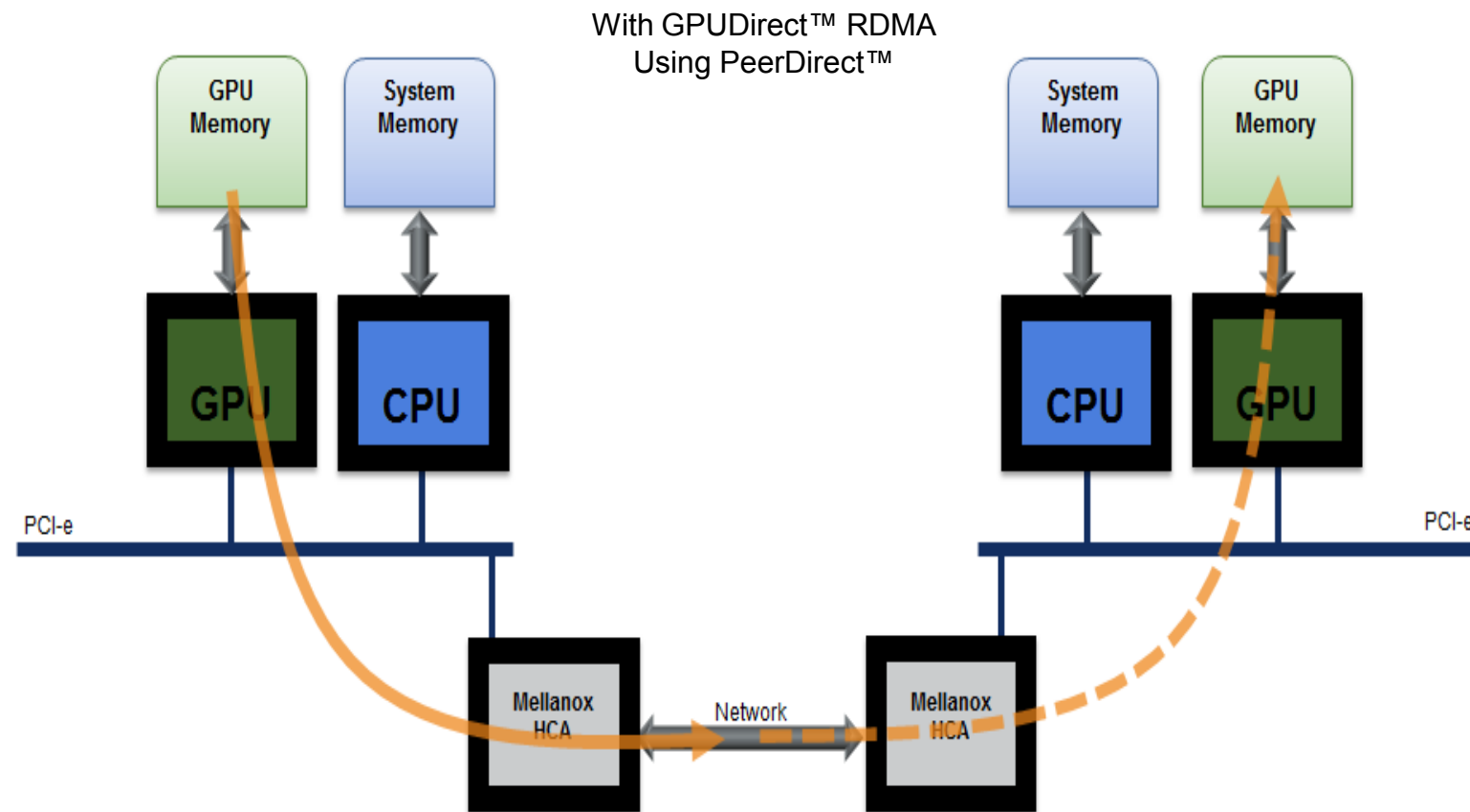


GPUDirect RDMA Technology

Maximize Performance via Accelerator and GPU Offloads

- Accelerated Communication With Network And Storage Devices
 - Avoid unnecessary system memory copies and CPU overhead by copying data directly to/from pinned third-party device memory
 - Peer-To-Peer Transfers Between third-party device and Mellanox RDMA devices
 - Use high-speed DMA transfers to copy data between P2P devices
 - Eliminate CPU bandwidth and latency bottlenecks using direct memory access (DMA)
- RDMA support
 - With PeerDirect™, memory of a third-party device can be used for Remote Direct Memory Access (RDMA) of the **data buffers** resulting in letting application run more efficiently.
 - Allow RDMA-based application to use a third-party device, such as a GPU for computing power, and RDMA interconnect at the same time w/o copying the data between the P2P devices.
 - Boost Message Passing Interface (MPI) Applications with zero-copy support
 - Support for RDMA transport over **InfiniBand and RoCE**

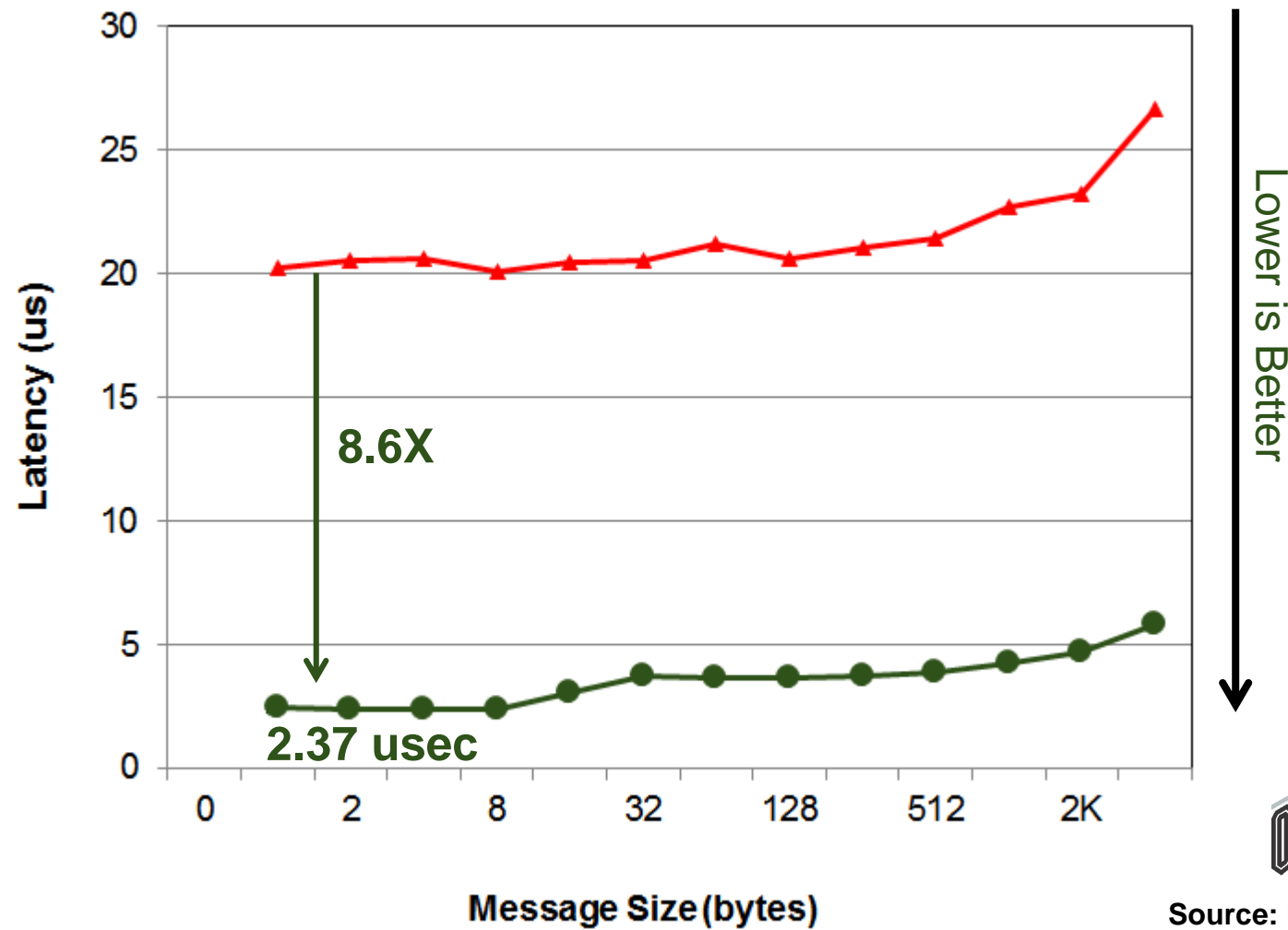
GPUDirect™ RDMA (GPUDirect 3.0)



- Eliminates CPU bandwidth and latency bottlenecks
- Uses remote direct memory access (RDMA) transfers between GPUs
- Resulting in significantly improved MPI efficiency between GPUs in remote nodes
- Based on PCIe PeerDirect technology

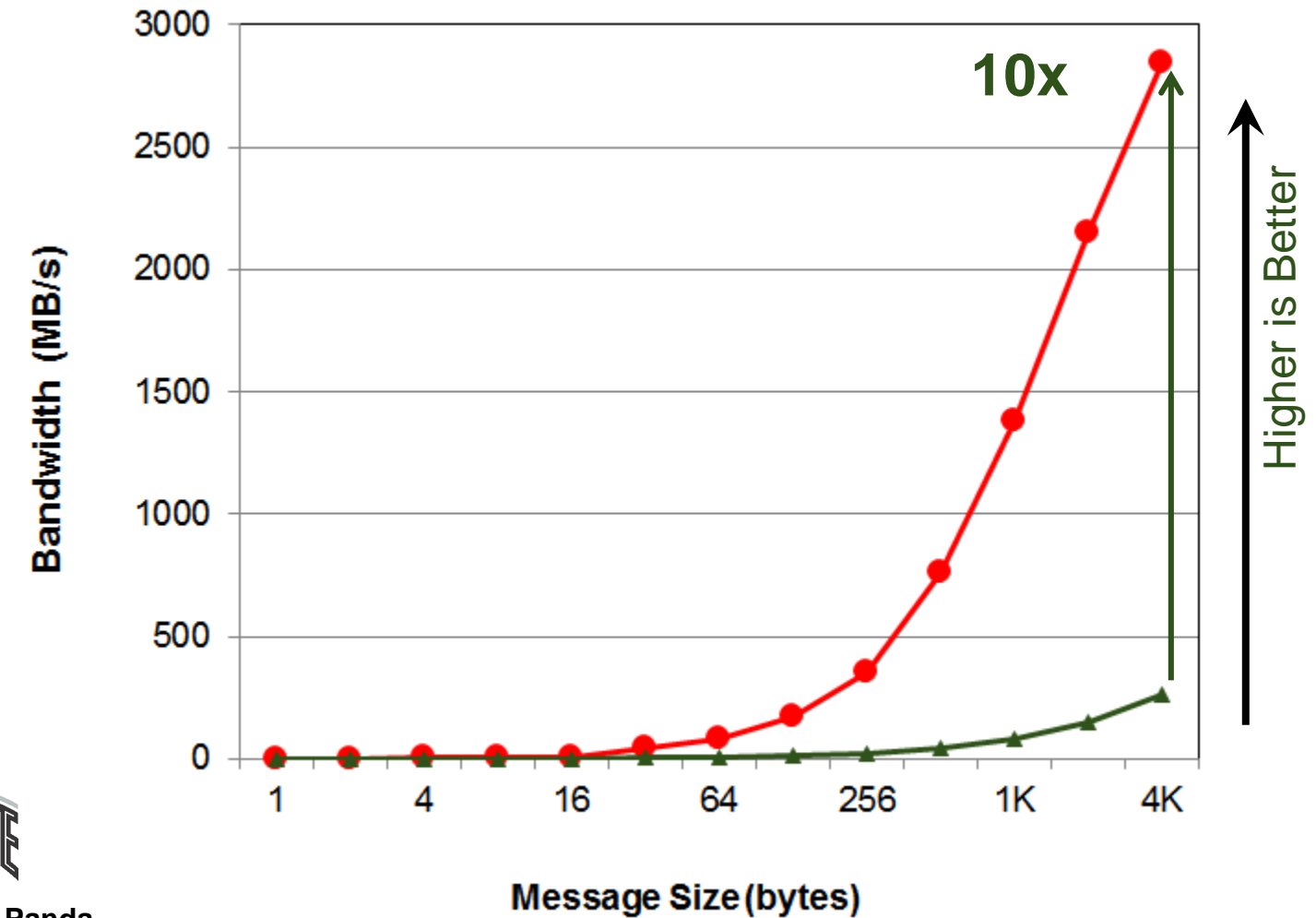
Performance of MVAPICH2 with GPUDirect RDMA

GPU-GPU Internode MPI Latency



Source: Prof. DK Panda

GPU-GPU Internode MPI Bandwidth

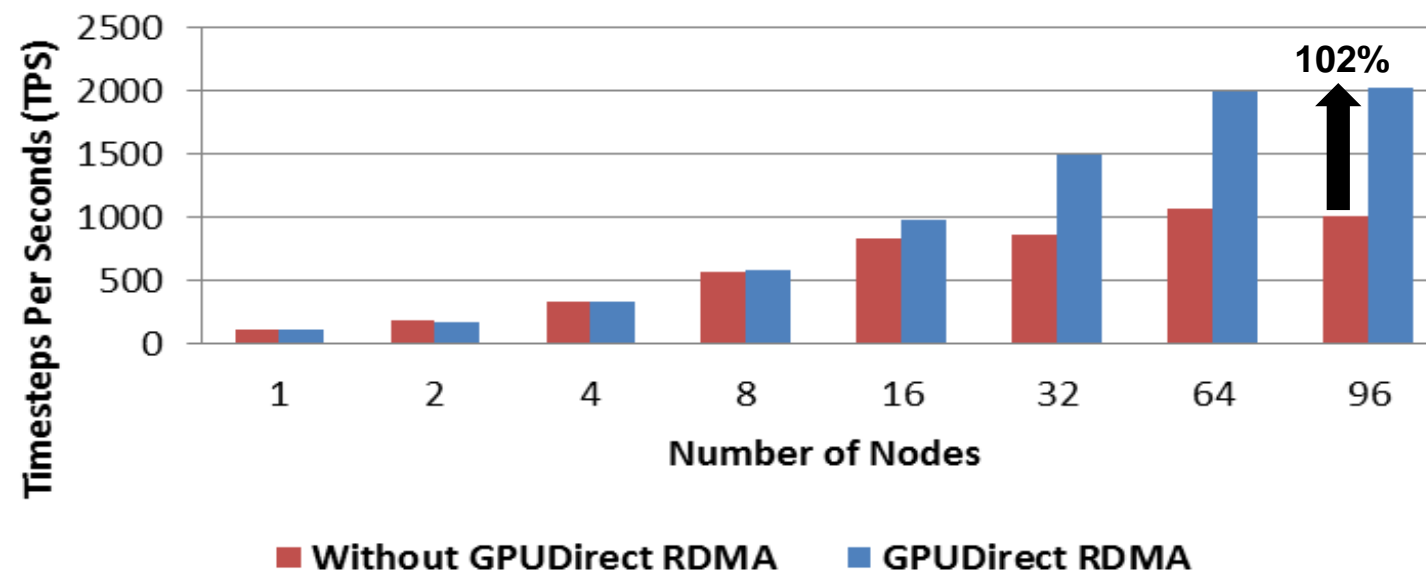


88% Lower Latency

10X Increase in Throughput

- HOOMD-blue is a general-purpose Molecular Dynamics simulation code accelerated on GPUs
- GPUDirect RDMA allows direct peer to peer GPU communications over InfiniBand
 - Unlocks performance between GPU and InfiniBand
 - This provides a significant decrease in GPU-GPU communication latency
 - Provides complete CPU offload from all GPU communications across the network
- Demonstrated up to 102% performance improvement with large number of particles

**HOOMD-blue Performance
(LJ Liquid Benchmark, 512K Particles)**



HOOMD
=blue

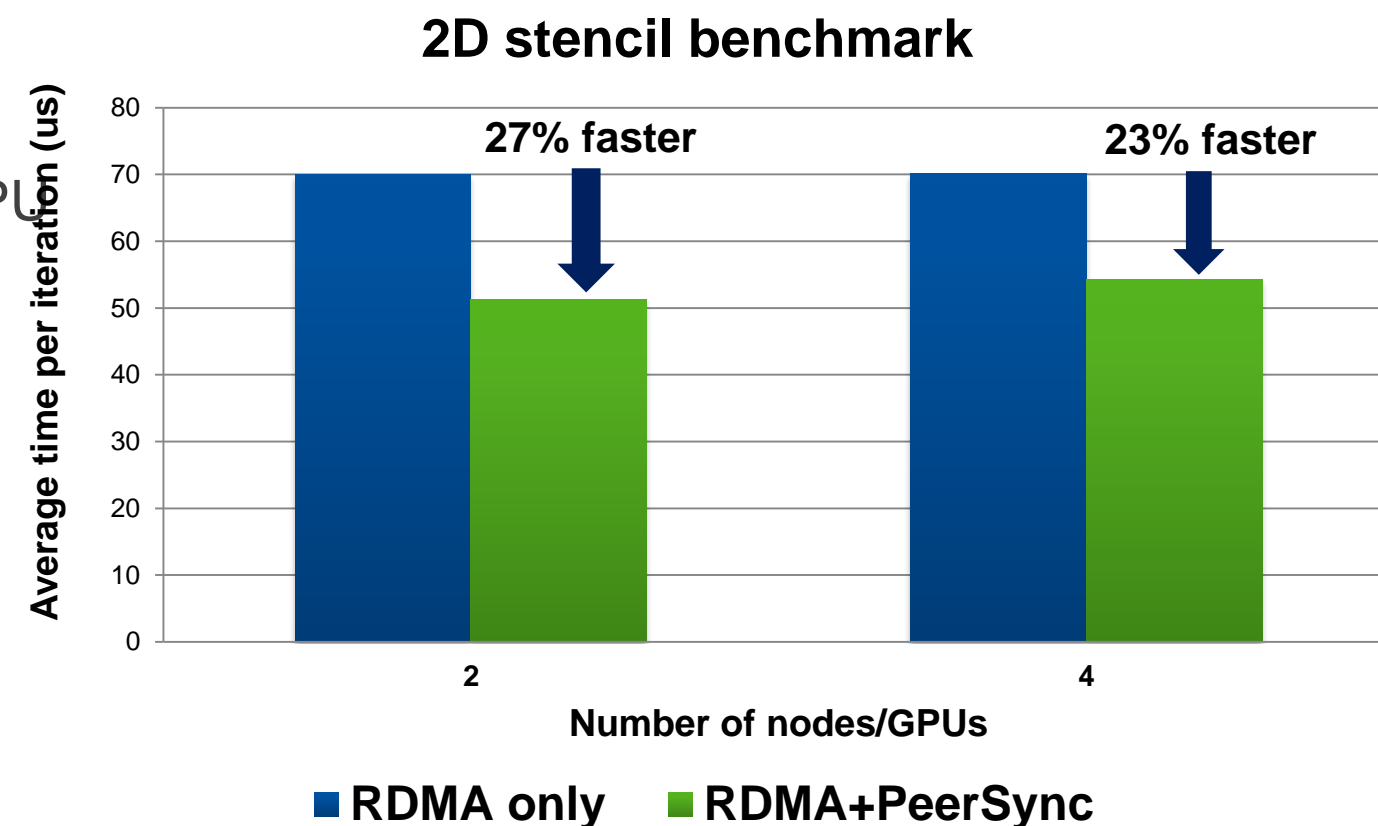
■ GPUDirect RDMA (3.0)

- direct data path between the GPU and Mellanox interconnect
- Control path still uses the CPU

■ GPUDirect ASync (GPUDirect 4.0)

- Both data path and control path go directly between the GPU and the Mellanox interconnect
- CPU prepares and queues communication tasks on GPU
- GPU triggers communication on HCA
- Mellanox HCA directly accesses GPU memory

**Maximum Performance
For GPU Clusters**





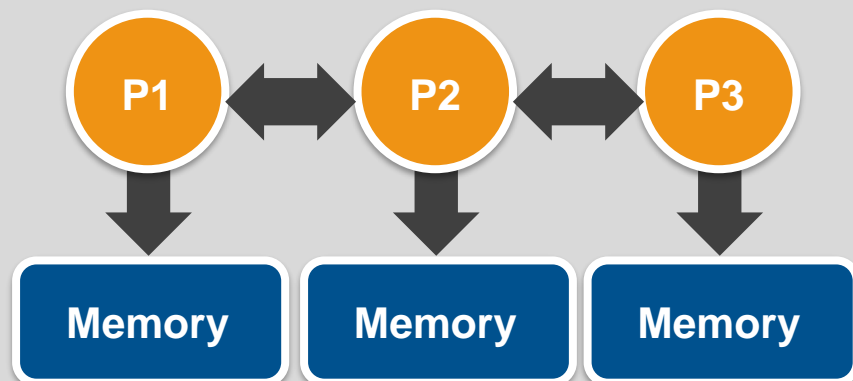
- Complete MPI, PGAS OpenSHMEM and UPC package
- Maximize application performance
- For commercial and open source applications
- Best out of the box experience



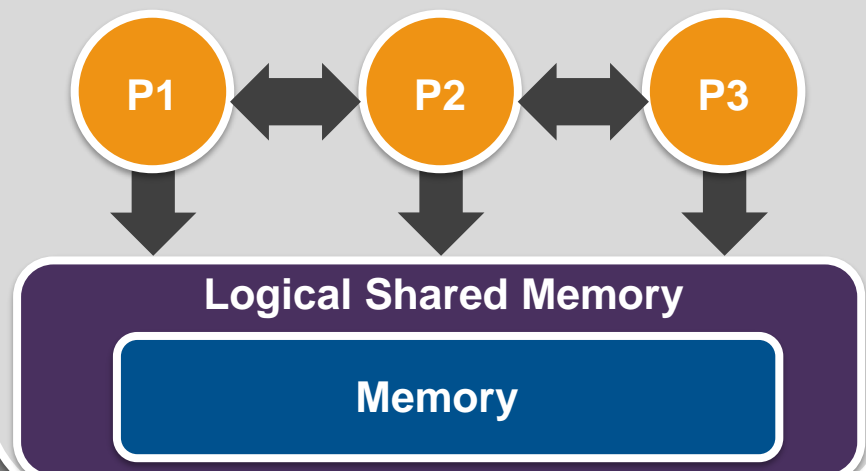
□ HPC-X – Mellanox Scalable HPC Toolkit

- Allow fast and simple deployment of HPC libraries
 - Both Stable & Latest Beta are bundled
 - All libraries are pre-compiled
 - Includes scripts/modulefiles to ease deployment
- Package Includes
 - OpenMPI/OpenSHMEM
 - BUPC (Berkeley UPC)
 - UCX
 - MXM
 - FCA-2.5
 - FCA-3.x (HCOLL)
 - KNEM
 - Allows fast intra-node MPI communication for large messages
 - Profiling Tools
 - Libibprof
 - IPM
 - Standard Benchmarks
 - OSU
 - IMB

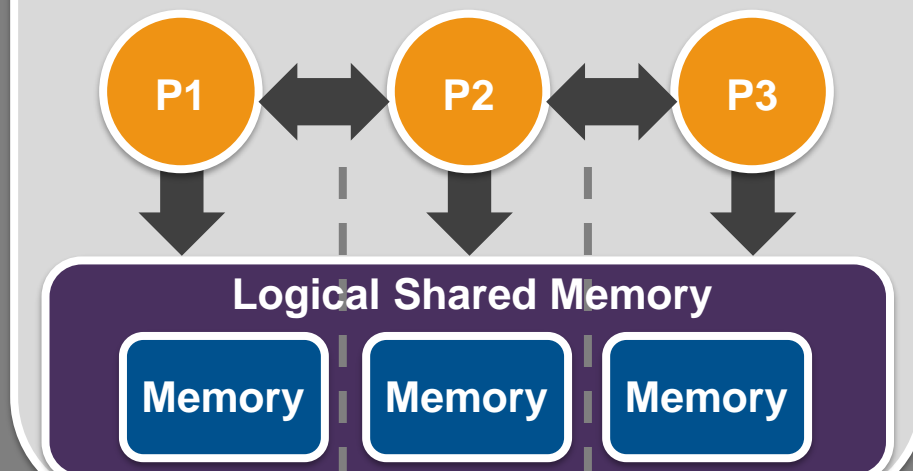
MPI



PGAS/SHMEM



PGAS/UPC



Point-to-Point: MXM -> UCX

- Reliable Messaging Optimized for Mellanox HCA
- Hybrid Transport Mechanism
- Efficient Memory Registration
- Receive Side Tag Matching

Collective: FCA

- Hardware Acceleration: SHArP, Multicast, CORE-Direct
- Topology Aware
- Separate Virtual Fabric for Collectives

InfiniBand Verbs API

To Install Mellanox HPC-X



- HPC-X Download Location
 - <http://www.mellanox.com/products/hpcx/>
- Download based on OS distribution and OFED version
- Support for Intel and GNU compilers for HPC-X for x86_64, power
- For more information:
 - http://www.mellanox.com/related-docs/prod_acceleration_software/HPC-X_Toolkit_User_Manual_v1.6.pdf
- Install HPC-X with no need for root privilege
 - \$ tar zxvf hpcx.tar

To Configure Mellanox HPC-X



- **Update environment variables using module**
 - `module use /opt/hpcx-v1.6.355-icc-MLNX_OFED_LINUX-3.1-1.1.0.1-redhat6.5-x86_64/modulefiles`
 - `module load hpcx`
- **Load KNEM Module on all compute nodes**
 - `# cd hpcx`
 - `# export HPCX_HOME=$PWD`
 - `# insmod $HPCX_HOME/knem/lib/modules/$(uname -r)/knem.ko`
 - `# chmod 666 /dev/knem`
- **FCA (v2.5)**
 - Prior to using FCA, the following command should be executed as root once on all cluster nodes
 - `# $HPCX_HOME/fca/scripts/udev-update.sh`
 - The FCA manager should be run on only one machine, and not on one of the compute nodes
 - `# $HPCX_HOME/fca/scripts/fca_managerd start`
- **FCA (v3.x) – HCOLL**
 - IPoIB addresses needed to be setup on the compute nodes

To Rebuild Mellanox HPC-X From Source





- Normally there is no need to recompile HPC-X
 - Open MPI and FCA/HCOLL and MXM libraries are included in case recompiling is needed
- Scenarios for rebuilding from source:
 - For threaded MPI library support – HPC-X is built without MPI thread multiple support (`--enable-thread`)
 - CUDA support (`--with-cuda`)
 - Support for other resource managers than SLURM
 - Compiler incompatibility (GNU and Intel library supported)
- To build HPC-X from source (from README):
 - `$ HPCX_HOME=/path/to/extracted/hpcx`
 - `$./configure --prefix=${HPCX_HOME}/hpcx-mpi \`
 `--with-knem=${HPCX_HOME}/knem \`
 `--with-fca=${HPCX_HOME}/fca --with-mxm=${HPCX_HOME}/mxm \`
 `--with-hcoll=${HPCX_HOME}/hcoll \`
 `--with-platform=contrib/platform/mellanox/optimized \`
 `--with-slurm --with-pmi $ make -j9 all && make -j9 install`

Mellanox Delivers Highest Applications Performance (HPC-X)



- Bull (Atos) testing results – Quantum Espresso application

				Intel MPI	Bull MPI (HPC-X)	
Quantum Espresso	Test Case	# nodes	time (s)	time (s)	Gain	
	A	43	584	368	37%	
	B	196	2592	998	61%	

Enabling Highest Applications Scalability and Performance

Collective Communication Library - HCOLL

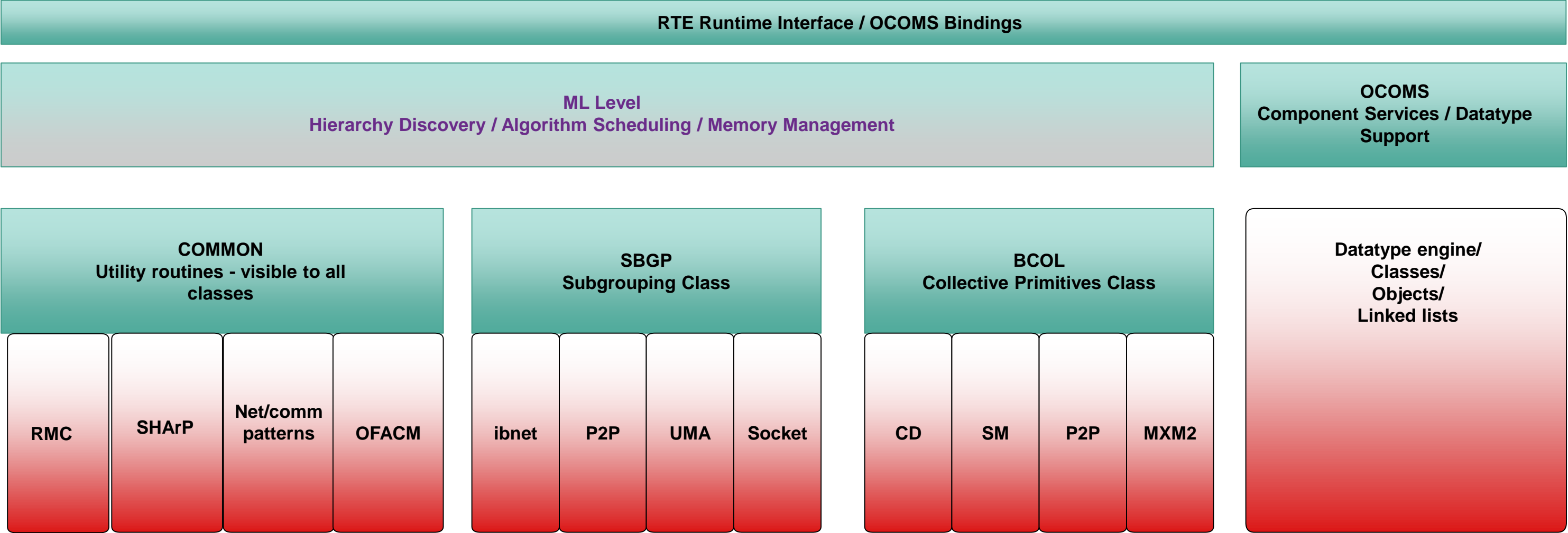
- Scalable infrastructure: Designed and implemented with current and emerging “*extreme-scale*” systems in mind
 - Scalable communicator creation
 - Scalable memory consumption
 - Scalable runtime interface
 - Asynchronous execution
- Flexible and Extensible: Plug-and-play component architecture
 - Leverage object-oriented design patterns
- Adaptive: Designed specifically for current and emerging heterogeneous memory subsystems
 - Can evolve gracefully in lockstep with new architectures
- Optimized collectives: Collective primitives are tuned to a particular communication substrate
- Expose CORE-Direct capabilities
 - Fully asynchronous, non-blocking collectives: Maximize the opportunity for the application developer to overlap computation with communication
 - Increase resilience to the effects of system noise on collective operations at extreme-scale
- Rapidly expose emerging Mellanox hardware features to ULPs with minimal effort e.g. multicast, DC, SHArP, UMR
- Easily integrated into other packages

- Blocking and non-blocking collective routines
- Modular component architecture
- Scalable runtime interface (RTE)
 - Successfully integrated into OMPI – “hcoll” component in “coll” framework
 - Successfully integrated in Mellanox OSHMEM
 - Experimental integration in MPICH
 - Working prototype SLURM/PMI2 plug-in
- Host level hierarchy awareness
 - Core groups*
 - Socket groups
 - UMA groups
- Support for network level topology awareness
 - Fat tree: switch-leaf subgroups
 - 3-D torus: subgroups of same torus dimension *
- Exposes Mellanox and InfiniBand specific capabilities
 - CORE-Direct
 - MXM 2(3).x
 - UD, RC, DC
 - UCX
 - Hardware multicast
 - SHArP
 - UMR for non-contiguous data*

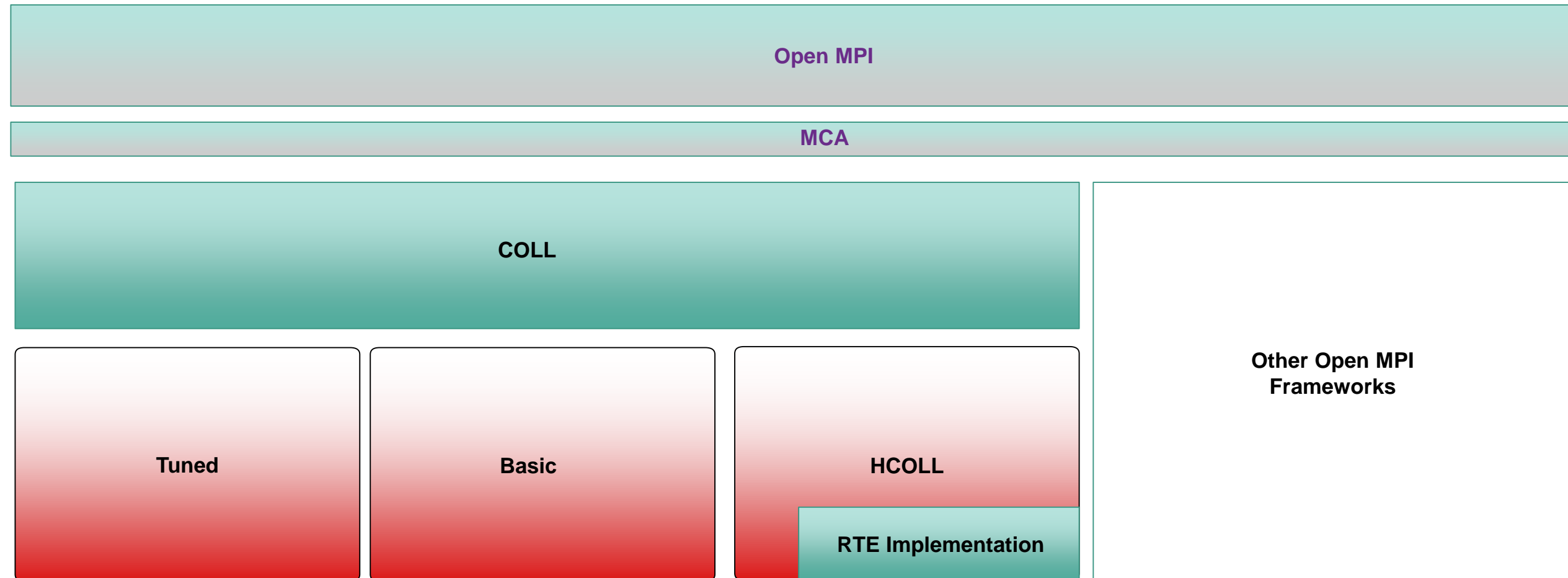
* To be released soon

■ Supported collectives in 3.0

- MPI_Allgather
- MPI_Allreduce
- MPI_Barrier
- MPI_Bcast
- MPI_Alltoallv
- MPI_Reduce
- MPI_Iallgather
- MPI_Iallreduce
- MPI_Ibarrier
- MPI_Ibcast
- MPI_Ireduce
- MPI_Ialltoallv
- MPI_Alltoall - Beta



Open MPI Integration: HCOLL component



■ FCA 2.5

- To enable FCA support
 - -mca coll_fca_enable 1
 - -mca coll_fca_np 0
 - -x fca_ib_dev_name=mlx5_0 (if multiple interfaces existed)

■ FCA 3.1+ (HCOLL)

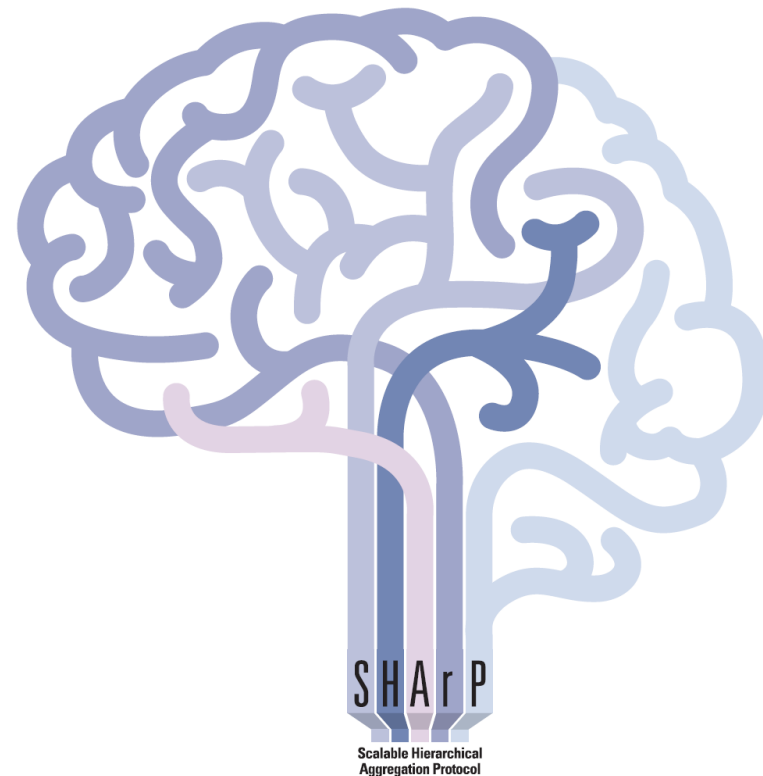
- To enable HCOLL support explicitly:
 - -mca coll_hcoll_enable 1
 - -mca coll_hcoll_np 0
 - -x HCOLL_MAIN_IB=mlx5_0:1 (if multiple interfaces existed)
- To enable HCOLL multicast support explicitly:
 - -x HCOLL_ENABLE_MCAST_ALL=1
 - -x HCOLL_MCAST_NP=0
- Comm context cache
 - -x HCOLL_CONTEXT_CACHE_ENABLE=1
- SHArP
 - -x HCOLL_ENABLE_SHARP=1

SHArP: Scalable Hierarchical Aggregation Protocol

In Network Data Reduction

Accelerating HPC Applications

- Significantly reduce MPI collective runtime
- Increase CPU availability and efficiency
- Enable communication and computation overlap

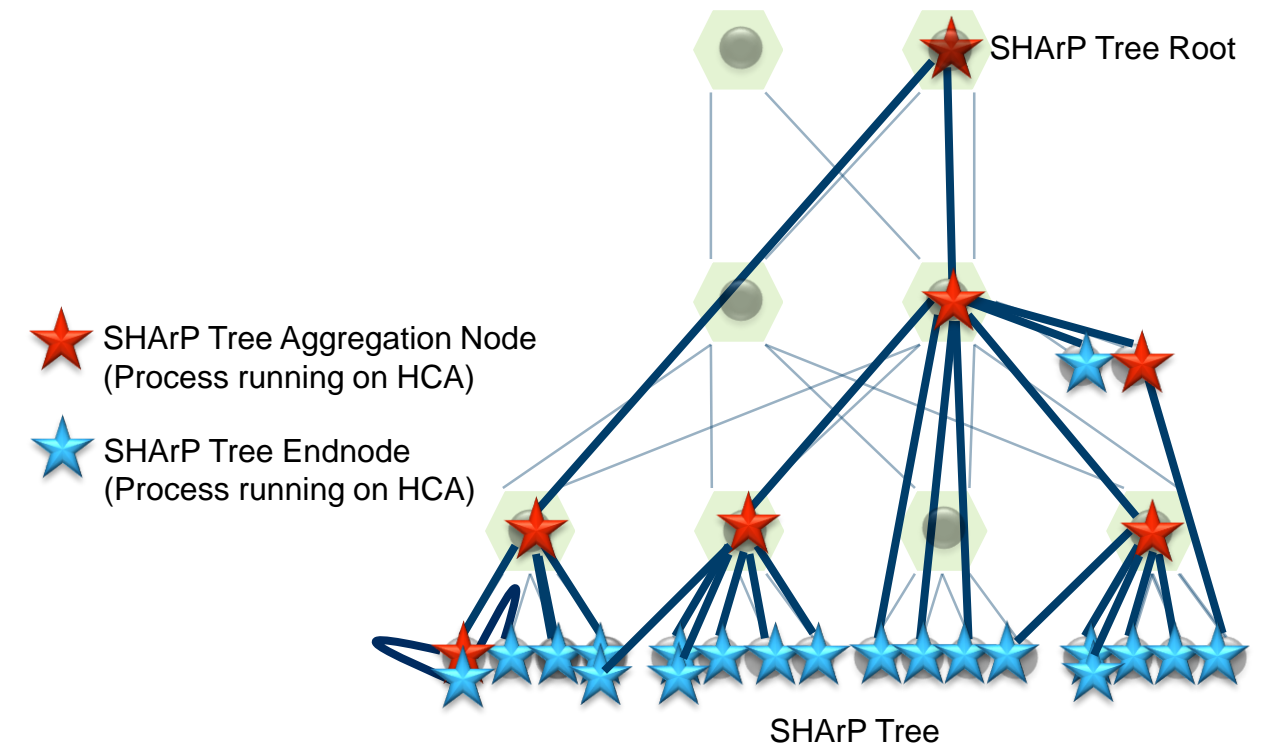


Enabling Artificial Intelligence Solutions to Perform Critical and Timely Decision Making

- Accelerating distributed machine learning
- Improving classification accuracy
- Reducing the number of batches needed for asynchronous training

SHArP: Scalable Hierarchical Aggregation Protocol

- **Reliable Scalable General Purpose Primitive**
 - In-network Tree based aggregation mechanism
 - Large number of groups
 - Multiple simultaneous outstanding operations
- **Applicable to Multiple Use-cases**
 - HPC Applications using MPI / SHMEM
 - Distributed Deep Learning applications
- **Scalable High Performance Collective Offload**
 - Barrier, Reduce, All-Reduce, Broadcast
 - Sum, Min, Max, Min-loc, max-loc, OR, XOR, AND
 - Integer and Floating-Point, 32 / 64 bit



SHArP
Scalable Hierarchical
Aggregation Protocol

■ Scalability Considerations

- Unlimited Group Size
- Large Number of Groups
- Hundreds of Simultaneous Outstanding Operations



■ Prerequisites

- SHArP software modules are delivered as part of HPC-X 1.6.392 or later
- Switch-IB 2 firmware - 15.1100.0072 or later
- MLXN OS - 3.6.1002 or later
- MLNX OpenSM 4.7.0 or later (available with MLNX OFED 3.3-x.x.x or UFM 5.6)

■ SHArP SW components:

- Libs
 - libsharp.so (low level api)
 - libsharp_coll.so (high level api)
- Daemons
 - sharpd, sharp_am
- Scripts
 - sharp_benchmark.sh
 - sharp_daemons_setup.sh
- Utilities
 - sharp_coll_dump_config
 - sharp_hello
 - sharp_mpi_test
- public API
 - sharp_coll.h

- **sharpd: SHArP daemon**
 - compute nodes
 - Light wait process
 - Almost 0% cpu usage
 - Only control path

- **sharp_am: Aggregation Manager daemon**
 - same node as Subnet Manager
 - Resource manager

SHArP: Configuring Subnet Manager

- Edit the opensm.conf file.
- Set the parameter “sharp_enabled” to “2”.
- Run OpenSM with the configuration file.
 - % opensm -F <opensm configuration file> -B
- Verify that the Aggregation Nodes were activated by the OpenSM, run "ibnetdiscover".

For example:

vendid=0x0

devid=0xcf09

sysimgguid=0x7cfe900300a5a2a0

caguid=0x7cfe900300a5a2a8

Ca 1 "H-7cfe900300a5a2a8" # "**Mellanox Technologies Aggregation Node**"

[1](7cfe900300a5a2a8) "S-7cfe900300a5a2a0"[37] # lid 256 lmc 0 "MF0;sharp2:MSB7800/U1" lid 512 4xFDR

SHArP: Configuring Aggregation Manager

- Create a configuration directory for the future SHArP configuration file.
 - % mkdir \$HPCX_SHARP_DIR/conf
- Create the fabric.lst file.
 - Copy the subnet LST file created by the Subnet Manager to the AM's configuration files directory
 - Rename it to fabric.lst :
 - % cp /var/log/opensm-subnet.lst \$HPCX_SHARP_DIR/conf/fabric.lst
- Create root GUIDs file.
 - Copy the root_guids.conf file if used for configuration of Subnet Manager to \$HPCX_SHARP_DIR/conf/root_guid.cfg (or)
 - Identify the root switches of the fabric and create a file with the node GUIDs of the root switches of the fabric.
 - For example : if there are two root switches files contains

```
0x0002c90000000001
0x0002c90000000008
```
- Create sharp_am.conf file

```
% cat > $HPCX_SHARP_DIR/conf/sharp_am.conf << EOF
fabric_lst_file $HPCX_SHARP_DIR/conf/fabric.lst
root_guids_file $HPCX_SHARP_DIR/conf/root_guid.cfg
ib_port_guid <PortGUID of the relevant HCA port or 0x0>
EOF
```

SHArP: Running SHArP Daemons



- setup the daemons

- `$HPCX_SHARP_DIR/sbin/sharp_daemons_setup.sh`

- Usage

- `sharp_daemons_setup.sh <-s SHArP location dir> <-r> <-d daemon> <-m>`
 - s - Setup SHArP daemon
 - r - Remove SHArP daemon
 - d - Daemon name (sharpd or sharp_am)
 - m - Add monit capability for daemon control
- `$HPCX_SHARP_DIR/sbin/sharp_daemons_setup.sh -s $HPCX_SHARP_DIR -d sharp_am`

SHArP: Running SHArP Daemons

■ sharp_am

- `$HPCX_SHARP_DIR/sbin/sharp_daemons_setup.sh -s $HPCX_SHARP_DIR -d sharp_am`
- Log : `/var/log/sharp_am.log`

■ Sharpd

- conf file: `$HPCX_SHARP_DIR/conf/sharpd.conf`
 - `ib_dev <relevant_hca:port>`
 - `sharpd_log_level 2`
- `$pdsh -w <hostlist> $HPCX_SHARP_DIR/sbin/sharp_daemons_setup.sh -s $HPCX_SHARP_DIR -d sharpd`
- Log : `/var/log/sharpd.log`

- Enabled through FCA-3.x (HCOLL)
- Flags
 - HCOLL_ENABLE_SHARP (default : 0)
 - 0 - Don't use SHArP
 - 1 - probe SHArP availability and use it
 - 2 - Force to use SHArP
 - 3 - Force to use SHArP for all MPI communicators
 - 4 - Force to use SHArP for all MPI communicators and for all supported collectives
 - HCOLL_SHARP_NP (default: 2)
 - Number of nodes(node leaders) threshold in communicator to create SHArP group and use SHArP collectives
 - SHARP_COLL_LOG_LEVEL
 - 0 – fatal , 1 – error, 2 – warn, 3 – info, 4 – debug, 5 – trace
 - HCOLL_BCOL_P2P_ALLREDUCE_SHARP_MAX
 - Maximum allreduce size run through SHArP

- Resources (quota)
 - SHARP_COLL_JOB_QUOTA_MAX_GROUPS
 - #communicators
 - SHARP_COLL_JOB_QUOTA_OSTS
 - Parallelism on communicator
 - SHARP_COLL_JOB_QUOTA_PAYLOAD_PER_OST
 - Payload/OST

- For complete list of SHARP COLL tuning options
 - \$HPCX_SHARP_DIR/bin/sharp_coll_dump_config -f

■ High level API

- sharp_coll.h
 - Public
 - Implementation of low level API
 - Matches with MPI semantics
 - libsharp_coll.so
 - Integrated into FCA-3.x

■ Low level API

- sharp.h
 - Not public yet. Will be soon
 - Finer grain control

■ JOB Init/finalize

- `int sharp_coll_init(struct sharp_coll_init_spec *sharp_coll_spec,
 struct sharp_coll_context **sharp_coll_context);`
- `int sharp_coll_finalize(struct sharp_coll_context *context);`

```
struct sharp_coll_init_spec {  
    uint64_t    job_id;                /**< Job unique ID */  
    int         world_rank;            /**< Global unique process number. */  
    int         world_size;            /**< Num of processes in the job. */  
    int         (*progress_func)(void); /**< External progress function. */  
    struct sharp_coll_config config;    /**< @ref sharp_coll_config "SHARP COLL Configuration". */  
    struct sharp_coll_out_of_band_colls oob_colls; /**< @ref sharp_coll_out_of_band_colls "List of OOB collectives". */  
};  
  
struct sharp_coll_out_of_band_colls {  
    int (*bcast) (void* context, void* buffer, int len, int root);  
    int (*barrier) (void* context);  
    int (*gather) (void * context, int root, void *sbuf, void *rbuf, int len);  
}
```

■ COMM Init/finalize

- `int sharp_coll_comm_init(struct sharp_coll_context *context,
 struct sharp_coll_comm_init_spec *spec,
 struct sharp_coll_comm **sharp_coll_comm);`
- `int sharp_coll_comm_destroy(struct sharp_coll_comm *comm);`

```
struct sharp_coll_comm_init_spec {  
    int    rank;                /**< Uniq process rank in the group. */  
    int    size;                /**< Size of the SHArP group. */  
    int    is_comm_world;       /**< Is universal group (MPI_COMM_WORLD). */  
    void   *oob_ctx;            /**< External group context for OOB functions. */  
};
```

■ Collective operations

- `int sharp_coll_do_barrier(struct sharp_coll_comm *comm);`
- `int sharp_coll_do_barrier_nb(struct sharp_coll_comm *comm,
 struct sharp_coll_request **handle);`
- `int sharp_coll_do_allreduce(struct sharp_coll_comm *comm,
 struct sharp_coll_reduce_spec *spec);`
- `int sharp_coll_do_allreduce_nb(struct sharp_coll_comm *comm,
 struct sharp_coll_reduce_spec *spec,
 struct sharp_coll_request **req);`

```
struct sharp_coll_reduce_spec {  
    int                root;                /**< [in] root rank number (ignored for allreduce) */  
    struct sharp_coll_data_desc sbuf_desc;  /**< [in] source data buffer desc */  
    struct sharp_coll_data_desc rbuf_desc;  /**< [out] destination data buffer desc */  
    enum sharp_datatype dtype;             /**< [in] data type @ref sharp_datatype */  
    int                length;              /**< [in] reduce operation size */  
    enum sharp_reduce_op op;                /**< [in] reduce operator @ref sharp_reduce_op */  
};
```


- SHArP (Sep, 2016)
 - Multiple node leaders
 - Result distribution over MCAST-UD
 - Group Trimming
 - Error flow
- New and improved CORE-Direct/Cross-Channel collectives
- UCX BCOL
- Full Datatypes support
- Power arch Optimizations

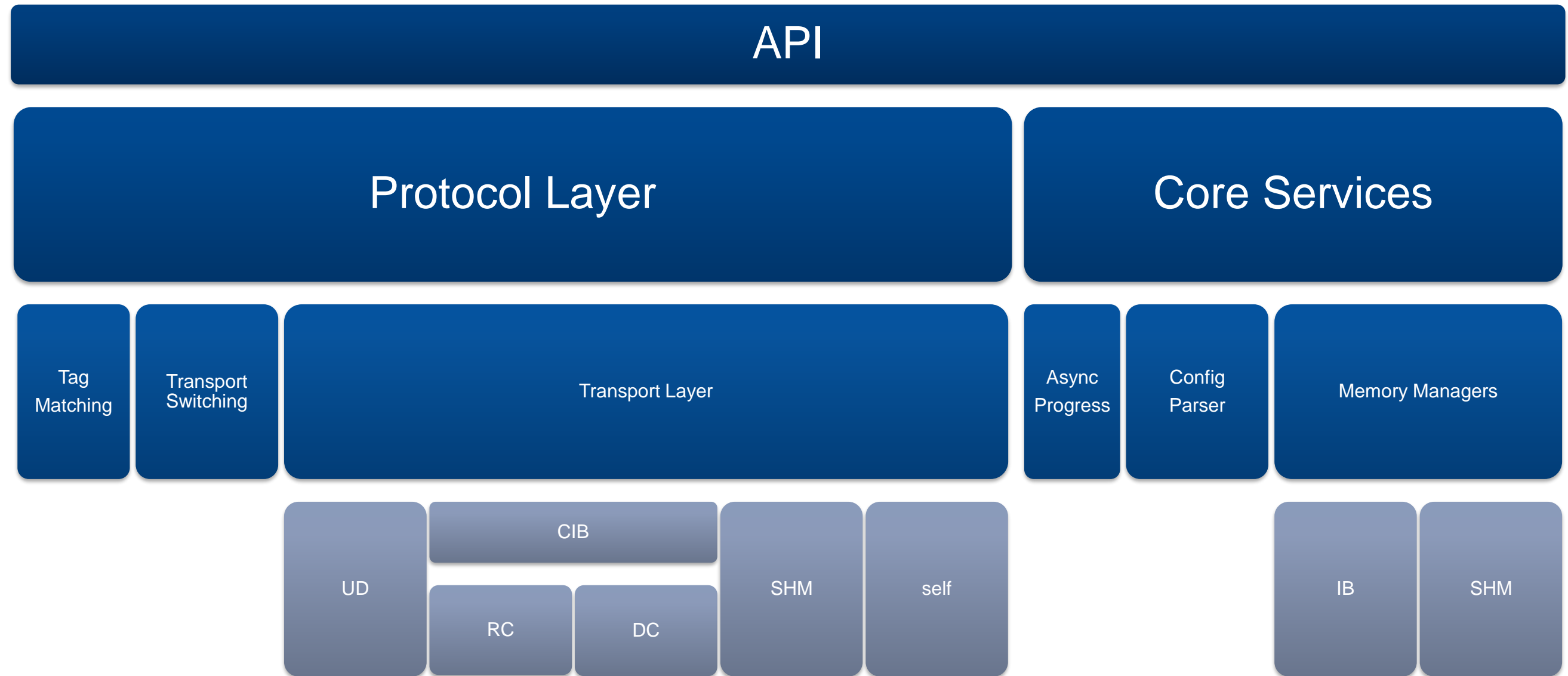
Point-to-Point Support

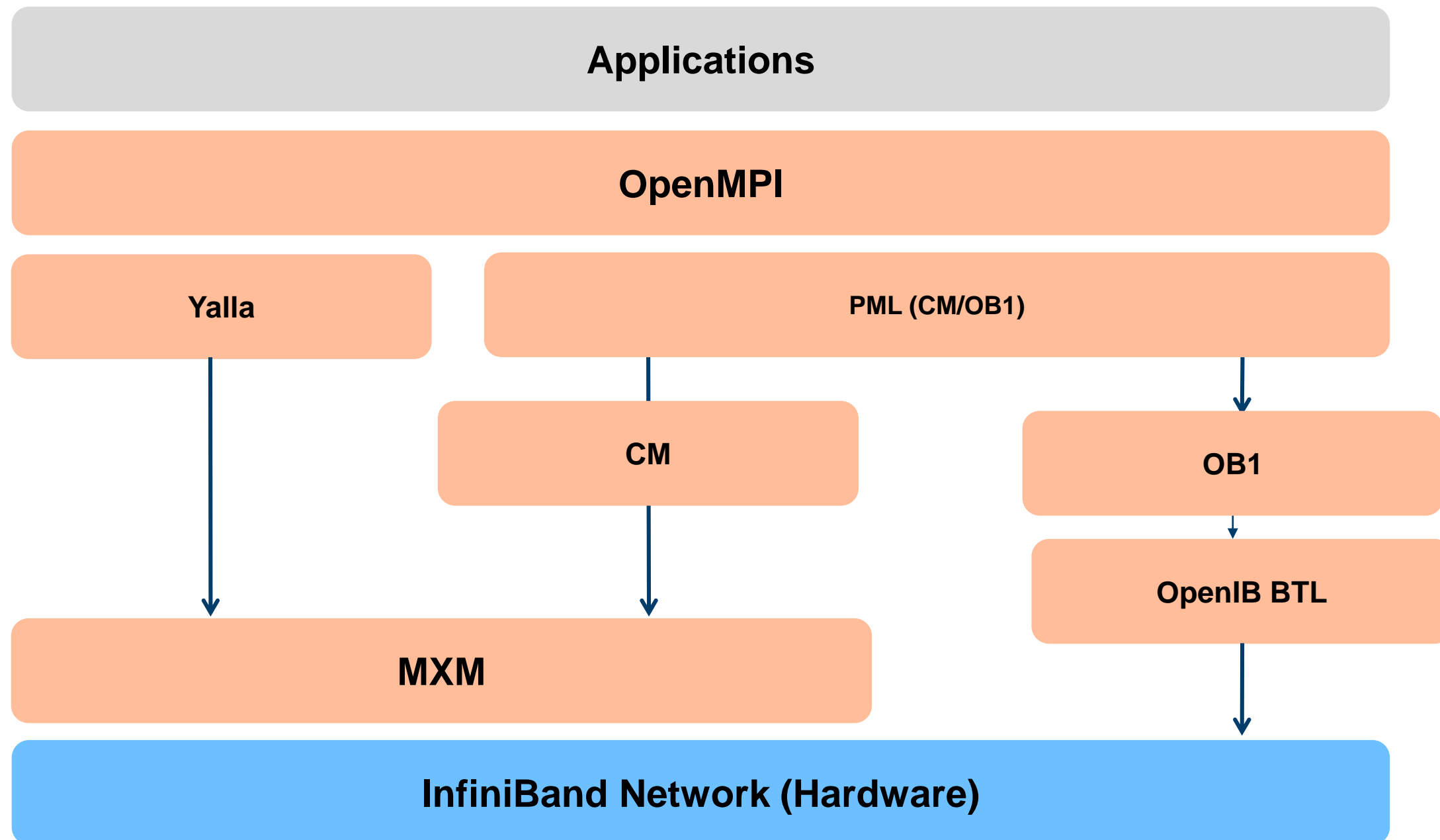
MXM and migrating over to UCX

MXM

- Point to Point acceleration
- InfiniBand and RoCE
- Ease of use - simple API
- Uses multiple transports
 - RC, UD, DC, SHM, loopback
- Hybrid mode – mix transports
 - switching between them as needed.
- Increases scalability by using DC and/or UD
- Efficient memory registration
- Improves shared memory communication using process-to-process memcpy (KNEM)
- Support for hardware atomics

- Thread-safe
 - Supports all MPI threading models
 - allow use of Hybrid model, i.e. MPI + OpenMP
- Re-use same protocols on different transports
 - Inline for small data
 - Fragmentation for medium data
 - Rendezvous for large data
- Scalability:
 - Fixed amount of buffers
 - Create connections on-demand
 - Reduce memory consumption per-connection
 - Scalable tag matching





□ Communications Library Support – MXM

➤ In OpenMPI v1.8, a new pml layer(yalla) was added that reduces overhead by bypassing layers and using the MXM library directly.

- for messages < 4K in size
 - ✓ Improves latency by 5%
 - ✓ Improves message rate by 50%
 - ✓ Improve bandwidth by up to 45%

HPC-X will choose this pml automatically

➤ To use this, pass the following command line in mpirun:

- `--mca pml yalla` (instead of `--mca pml cm --mca mtl mxm`)

■ MXM:

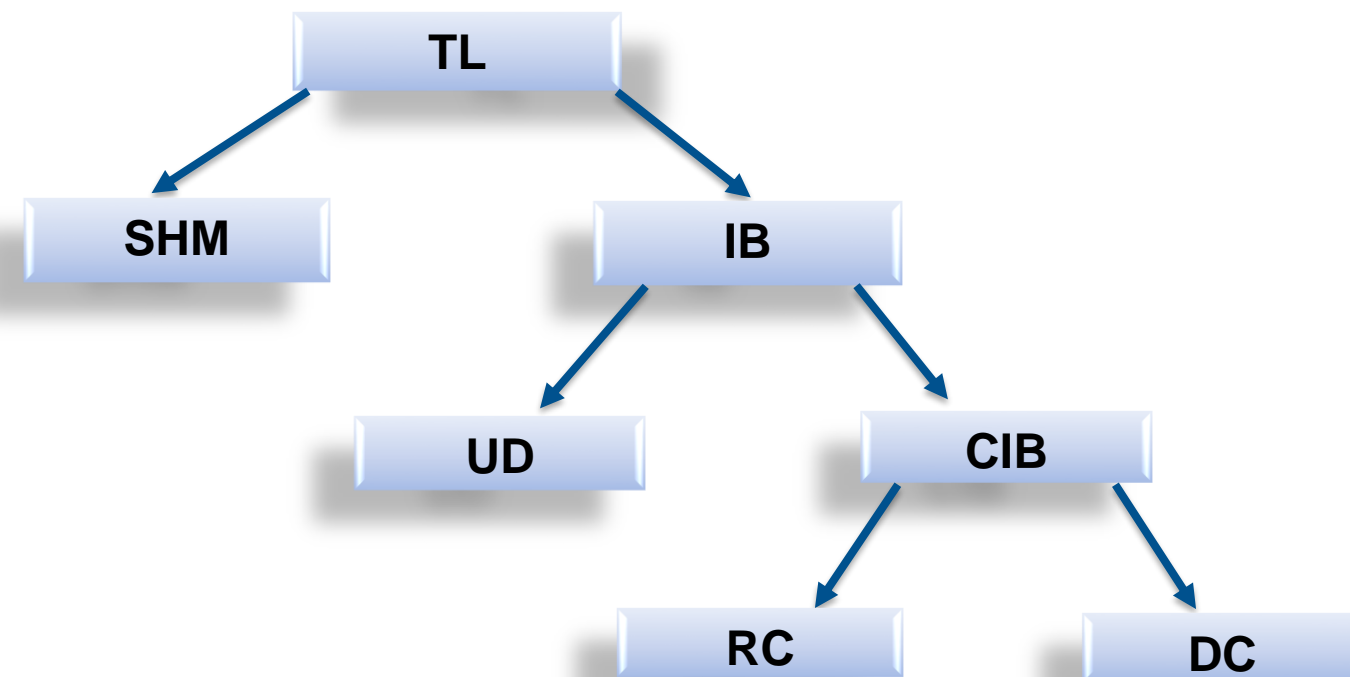
- To enable MXM support explicitly:
 - -mca pml yalla
- Transport selection
 - -x MXM_TLS=self,shm,**ud**
- Device selection
 - -x MXM_RDMA_PORTS=mlx5_0:1 (if multiple interfaces existed)
- For running low-level micro benchmarks, use RC transport
 - -x MXM_TLS=self,shm,**rc**
- To disable MXM and enable openib BTL
 - --mca pml ob1

MXM parameters hierarchy

- Each class has its own set of parameters.
- Each parameter from the parent class defines the same one in all descendant classes.
- For example MXM_**IB**_RX_QUEUE_LEN=1024 defines the same RX queue length in UD, RC and DC.
- But if we set the parameter together with MXM_**RC**_RX_QUEUE_LEN=2048 → UD and DC components RX queues will have length of 1024 and 2048 in RC.

To see the all list of parameters with their defaults:

```
cd HPC-X-INSTALL-DIR/mxm/bin  
./mxm_dump_config
```



Each param has **MXM_TL_** prefix

The next set of parameters could be useful for big data transactions tuning:

MXM_TL_RNDV_THRESH - Threshold for using rendezvous protocol.
Smaller value may harm performance,
but too large value can cause a deadlock in the application.
Default is 16384.

- ❑ Stands for “shared memory”
- ❑ Used for inter-process communication in the same node
- ❑ Works out of the box
- ❑ KNEM DMA will be automatically selected by MXM

Each param has **MXM_IB_** prefix

The next two ones could be useful for small messages rate increasing:

MXM_IB_CQ_MODERATION - Number of send WQEs for which completion is requested, the default is 64.

MXM_IB_TX_BATCH - Number of send WREs to batch in one post-send list.

Larger values reduce the CPU usage, but increase the latency and pipelining between sender and receiver, the default is 16.

Use the next parameter in order to run your job over ROCE:

MXM_IB_GID_INDEX - GID index to use as local Ethernet port address, 0 is the default value.

- ✓ Scalable
- ✓ Can be used with AR
- ✓ ROCE support
- ✓ Protocols: RNDV, ZCOPY
- ✓ Each param has **MXM_UD_** prefix
- ✓ Each send will be fragmented on sequence of MTUs sends
- ✓ Defaults:
 - MXM_UD_RNDV_THRESH=256k;**
 - MXM_UD_HARD_ZCOPY_THRESH=64k;**

UD - some useful parameters



MXM_UD_WINDOW_SIZE - How many un-acknowledged packets can be on the fly,
the default is 1024.

MXM_UD_ACK_TIMEOUT - Timeout for getting an acknowledgment for sent packet,
the default is 300ms.

UD RNDV ZCOPY



- Good for big buffers when *memcpy* from internal to user buffer could be an issue

We can manage UD RNDV ZCOPY protocol with the next parameters:

MXM_UD_RNDV_ZCOPY_WIN_SIZE - UD RNDV ZCOPY window size, the default is 1024.

MXM_UD_RNDV_ZCOPY_NUM_QPS - The number of UD QPs for RNDV ZCOPY protocol, the default is 64.

MXM_UD_RNDV_ZCOPY_WIN_TIMEOUT - Timeout for getting window acknowledgment from a remote peer, the default is "1800ms".

MXM_UD_RNDV_ZCOPY - Enable UD RNDV ZCOPY protocol (y/n), the default is "y".

Need to activate the Adaptive Routing at the Subnet Management (SM) level.

MXM configuration parameter -x **MXM_UD_FIRST_SL=8** (where SM has been configured to use SL=8 for adaptive routing)

Examples



OSU MPI All-to-All Latency Test - 32 procs on 2 nodes with UD

MXM_UD_RNDV_ZCOPY=n

# Size	Avg Latency(us)
1	18.48
2	18.19
4	18.41
8	19.00
16	22.10
32	25.59
64	31.02
128	46.24
256	68.14
512	139.26
1024	189.07
2048	400.34
4096	581.21
8192	966.42
16384	1781.00
32768	3398.96
65536	6198.17
131072	12656.01
262144	25729.65
524288	50504.39
1048576	99179.86
2097152	195143.01
4194304	388099.91

MXM_UD_RNDV_ZCOPY=y
MXM_UD_RNDV_ZCOPY_WIN_SIZE=1024

# Size	Avg Latency(us)
1	18.16
2	18.25
4	18.35
8	18.96
16	22.15
32	25.38
64	30.61
128	46.20
256	67.70
512	139.68
1024	188.45
2048	399.74
4096	581.53
8192	964.70
16384	1793.53
32768	3390.14
65536	6210.84
131072	12664.36
262144	19489.20
524288	38133.77
1048576	75860.85
2097152	151774.91
4194304	303383.81

Examples



OSU MPI Multiple Bandwidth / Message Rate Test - 32 procs on 2 nodes with UD

Defaults

# Size	MB/s	Messages/s
1	18.03	18026624.82
2	24.59	12294717.05
4	49.45	12362597.59
8	97.66	12208049.47
16	195.17	12197821.48
32	386.50	12078106.99
64	736.31	11504845.35
128	1148.48	8972464.40
256	1528.40	5970321.64
512	2141.69	4182988.69
1024	3367.09	3288170.57
2048	6178.14	3016668.92
4096	5687.36	1388516.30
8192	6121.58	747263.71
16384	6257.47	381925.41
32768	6110.82	186487.47
65536	5956.91	90895.21
131072	6018.24	45915.55
262144	15509.30	59163.27
524288	15600.46	29755.52
1048576	15702.19	14974.78
2097152	15757.31	7513.67
4194304	15796.38	3766.15
8388608	15802.43	1883.80
16777216	15805.43	942.08
33554432	15802.25	470.94

MXM_UD_HARD_ZCOPY_THRESH=32k
MXM_UD_RNDV_THRESH=32k

# Size	MB/s	Messages/s
1	14.96	14960286.31
2	25.07	12535704.18
4	48.55	12138569.53
8	96.11	12014230.60
16	197.82	12363509.51
32	405.37	12667675.01
64	695.73	10870772.04
128	1148.90	8975745.81
256	1562.98	6105402.21
512	2148.77	4196821.99
1024	3341.03	3262721.68
2048	6046.14	2952218.83
4096	5747.08	1403096.90
8192	6115.85	746563.76
16384	6277.95	383175.48
32768	13576.23	414313.56
65536	14964.31	228337.25
131072	15391.85	117430.48
262144	15541.13	59284.71
524288	15600.30	29755.21
1048576	15714.73	14986.74
2097152	15765.52	7517.58
4194304	15797.68	3766.46
8388608	15807.84	1884.44
16777216	15811.52	942.44
33554432	15802.17	470.94

UD Adaptive Routing – All-to-All 128 Haswell nodes, 28 PPN, FDR adapters, EDR network, 1:4 blocking



Message Size (bytes)	Latency – NO Adaptive Routing (usec)	Latency – with adaptive routing (usec)	Percent Improvement using Adaptive Routing
64	13413.2	12429.4	7.334699
128	19725.8	17397.1	11.80536
256	33157.6	27360.1	17.48449
512	68007.4	56606.8	16.76381
1024	75764.8	56596.0	25.30042
2048	89690.0	82247.3	8.298242
4096	170533.8	151588.2	11.10961
8192	335020.0	294236.3	12.17351
16384	669354.9	580389.7	13.29118
32768	1343660.5	1166258.8	13.20287
65536	2714040.4	2364910.2	12.86385
131072	5436677.8	4761931.9	12.411

The best choice when the amount of procs is not big.

The Rendezvous protocol is based on RDMA (*READ / WRITE* – use **MXM_RC_RNDV_MODE** param) capabilities.

Some defaults:

MXM_RC_MSS = 4224B

MXM_RC_HARD_ZCOPY_THRESH = 16k

MXM_RC_RX_QUEUE_LEN = 16000

MXM_RC_TX_MAX_BUFS = -1 i.e. infinite value, it's strongly recommended to not change this param.

Receive queue resizing:

MXM_RC_RX_SRQ_FILL_SIZE – receive buffers to pre-post, the value will be increased after each async SRQ limit event up to **MXM_RC_RX_QUEUE_LEN**,
the default is 2048.

MXM_RC_RX_SRQ_RESIZE_FACTOR - For each SRQ watermark event it will be resized by this factor,
the default is “4.0”.

EAGER RDMA protocol



Use the next set of parameters for managing:

MXM_RC_USE_EAGER_RDMA - Use RDMA WRITE for small messages (y/n), the default is “y”.

MXM_RC_EAGER_RDMA_THRESHOLD - Use RDMA for short messages after this number of messages are received from a given peer, the default is 16.

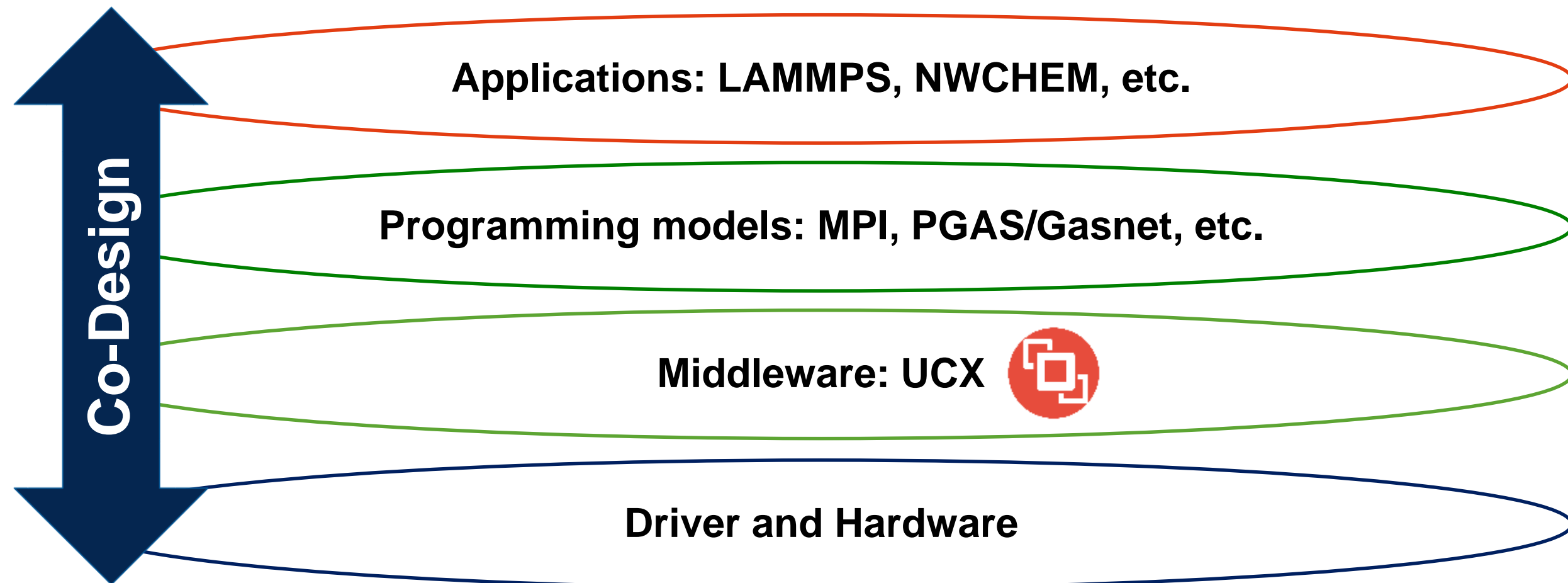
MXM_RC_MAX_RDMA_CHANNELS - Maximum number of peers allowed to use RDMA for short messages, the default is 8.

MXM_RC_EAGER_RDMA_BUFFS_NUM - Number of RDMA buffers to allocate per rdma channel, the default is 32.

MXM_RC_EAGER_RDMA_BUFF_LEN - Maximum size (in bytes) of eager RDMA messages, the default is 4224 (like MSS default).

UCX

- Collaboration between industry, laboratories, and academia
- To create open-source production grade communication framework for data centric and HPC applications
- To enable the highest performance through co-design of software-hardware interfaces



Co-Design Effort Between National Laboratories, Academia, and Industry

MXM

- Developed by Mellanox Technologies
- HPC communication library for InfiniBand devices and shared memory
- Primary focus: MPI, PGAS

PAMI

- Developed by IBM on BG/Q, PERCS, IB VERBS
- Network devices and shared memory
- MPI, OpenSHMEM, PGAS, CHARM++, X10
- C++ components
- Aggressive multi-threading with contexts
- Active Messages
- Non-blocking collectives with hw acceleration support

UCCS

- Developed by ORNL, UH, UTK
- Originally based on Open MPI BTL and OPAL layers
- HPC communication library for InfiniBand, Cray Gemini/Aries, and shared memory
- Primary focus: OpenSHMEM, PGAS
- Also supports: MPI

**UCX is an Integration of Industry
and Users Design Efforts**

A Collaboration Effort



- Mellanox co-designs network interface and contributes MXM technology
 - Infrastructure, transport, shared memory, protocols, integration with OpenMPI/SHMEM, MPICH



- ORNL co-designs network interface and contributes UCCS project
 - InfiniBand optimizations, Cray devices, shared memory



- NVIDIA co-designs high-quality support for GPU devices
 - GPUDirect, GDR copy, etc.



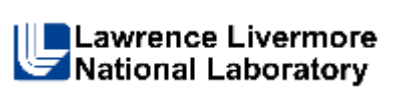
- IBM co-designs network interface and contributes ideas and concepts from PAMI



- UH/UTK focus on integration with their research platforms



- ANL focus on UCX CH4 support



- Collaboration between industry, laboratories, and academia
- Create open-source production grade communication framework for HPC applications
- Enable the highest performance through co-design of software-hardware interfaces
- Unify industry - national laboratories - academia efforts

API

Exposes broad semantics that target data centric and HPC programming models and applications

Performance oriented

Optimization for low-software overheads in communication path allows near native-level performance

Production quality

Developed, maintained, tested, and used by industry and researcher community

Community driven

Collaboration between industry, laboratories, and academia

Research

The framework concepts and ideas are driven by research in academia, laboratories, and industry

Cross platform

Support for Infiniband, Cray, various shared memory (x86-64 and Power), GPUs

Co-design of Exascale Network APIs

UC-S for Services

This framework provides basic infrastructure for component based programming, memory management, and useful system utilities

Functionality:

Platform abstractions, data structures, debug facilities.

UC-T for Transport

Low-level API that expose basic network operations supported by underlying hardware. Reliable, out-of-order delivery.

Functionality:

Setup and instantiation of communication operations.

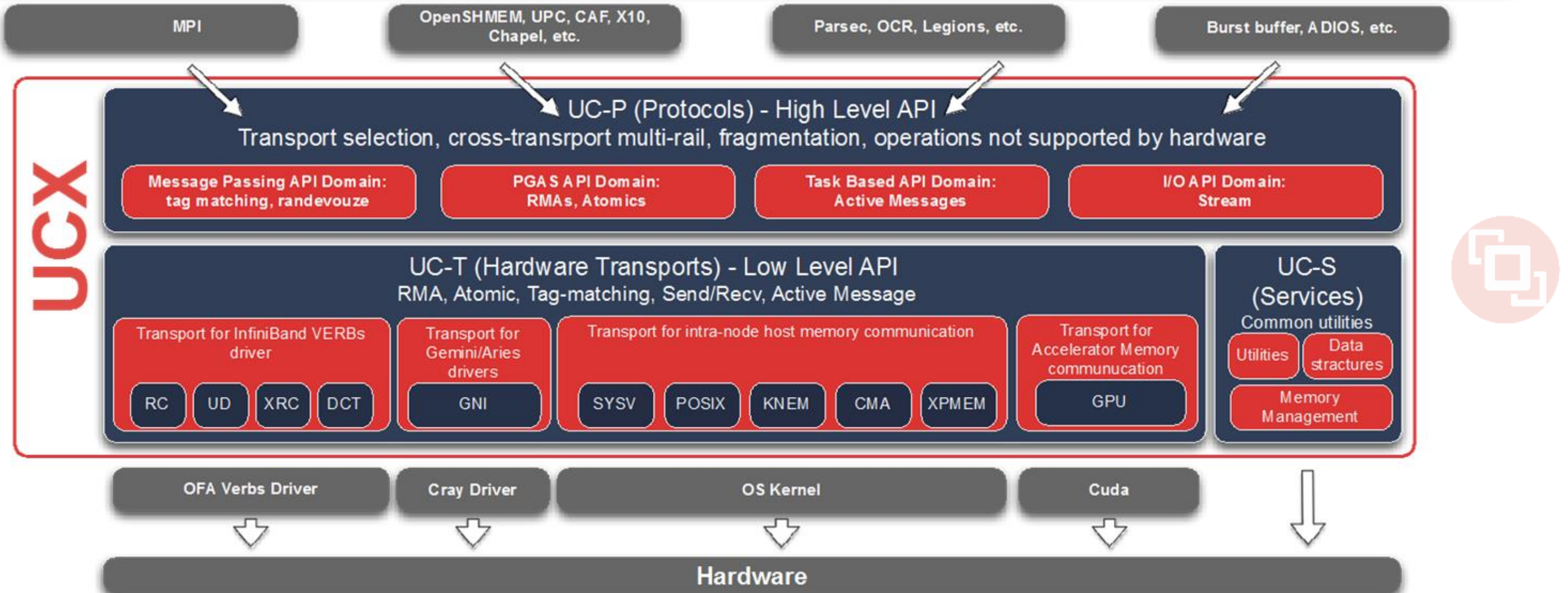
UC-P for Protocols

High-level API uses UCT framework to construct protocols commonly found in applications

Functionality:

Multi-rail, device selection, pending queue, rendezvous, tag-matching, software-atomics, etc.

Applications



Unified, Light-Weight, High-Performance Communication Framework

Differences between UCX and MXM

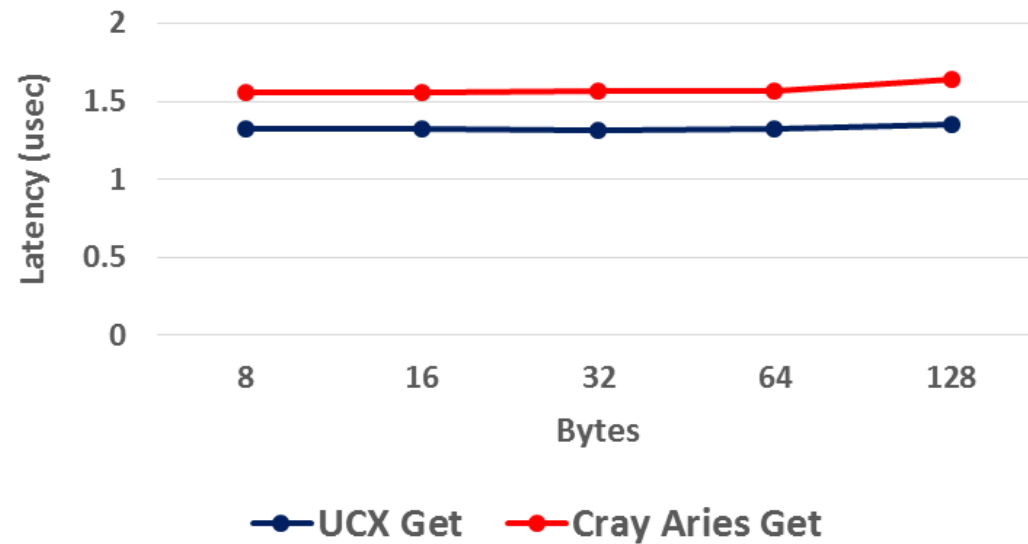


- Simple and consistent API
- Choosing between low-level and high-level API allows easy integration with a wide range of applications and middleware.
- Protocols and transports are selected by capabilities and performance estimations, rather than hard-coded definitions.
- Support thread contexts and dedicated resources, as well as fine-grained and coarse-grained locking.
- Accelerators are represented as a transport, driven by a generic “glue” layer, which will work with all communication networks.
- Usage: add `--mca pml ucx` to your OMPI run command line

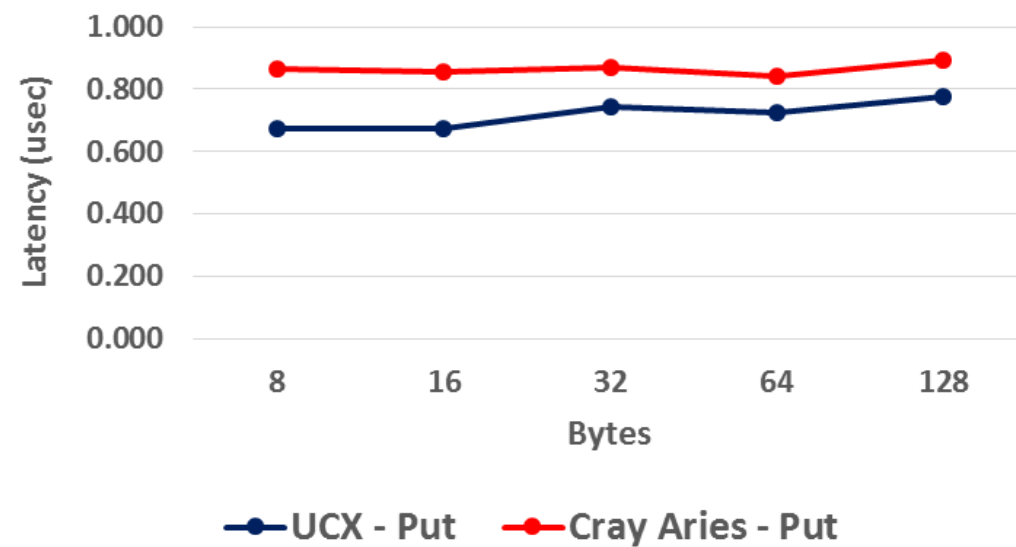
HPC-X UCX Performance



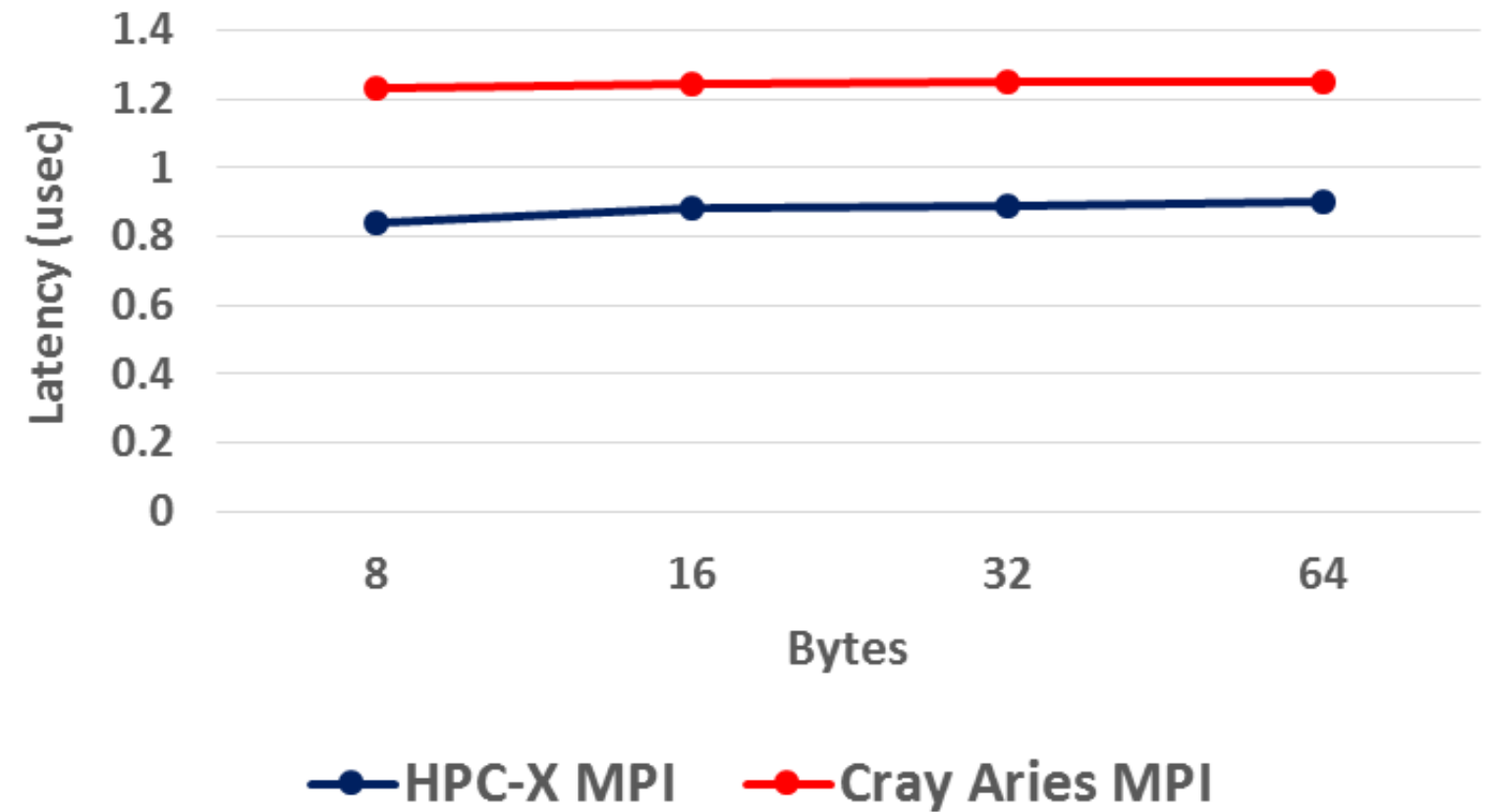
GET Latency



PUT Latency



MPI Latency



- **uct_worker_h**

A context for separate progress engine and communication resources. Can be either thread-dedicated or shared.

- **uct_pd_h** (will be renamed to **uct_md_h**)

Memory registration domain. Can register user buffers and allocate registered memory.

- **uct_iface_h**

Communication interface, created on a specific memory domain and worker. Handles incoming active messages and spawns connections to remote interfaces.

- **uct_ep_h**

Connection to a remote interface. Used to initiate communications.

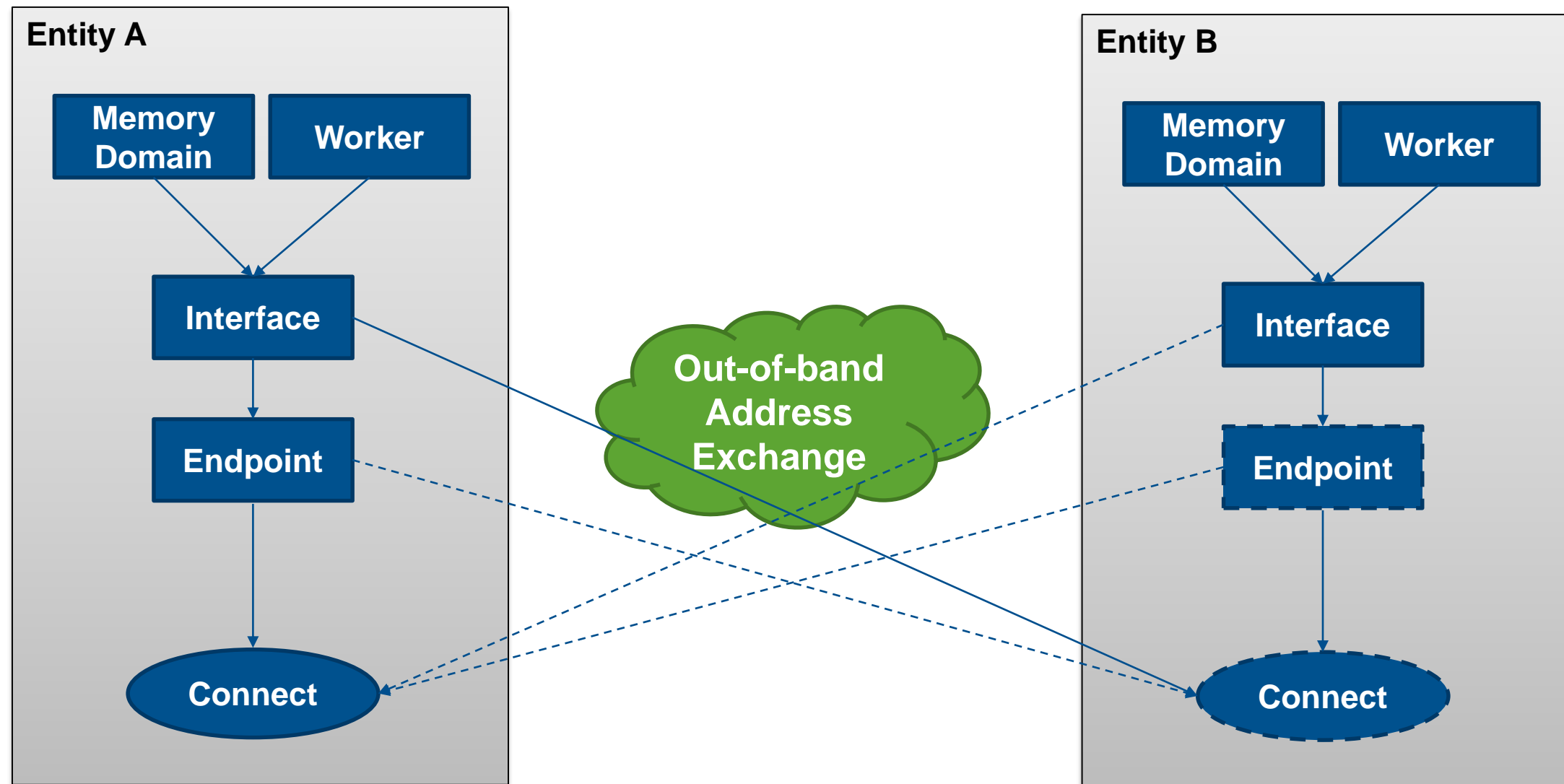
- **uct_mem_h**

Registered memory handle.

- **uct_rkey_t**

Remote memory access key.

UCT initialization



- Register memory within the domain
 - Can potentially use a cache to speedup registration
- Allocate registered memory.
- Pack memory region handle to a remote-key-buffer
 - Can be sent to another entity.
- Unpack a remote-key-buffer into a remote-key
 - Can be used for remote memory access.

- Not everything has to be supported.
 - Interface reports the set of supported primitives.
 - UCP uses this info to construct protocols.
- Send active message (active message id)
- Put data to remote memory (virtual address, remote key)
- Get data from remote memory (virtual address, remote key)
- Perform an atomic operation on remote memory:
 - Add
 - Fetch-and-add
 - Swap
 - Compare-and-swap
- Insert a fence
- Flush pending communications

- UCT communications have a size limit
 - Interface reports max. allowed size for every operation.
 - Fragmentation, if required, should be handled by user / UCP.
- Several data “classes” are supported:
 - “short” – small buffer.
 - “bcopy” – a user callback which generates data (in many cases, “memcpy” can be used as the callback).
 - “zcopy” – a buffer and it’s memory region handle. Usually large buffers are supported.
- Atomic operations use a 32 or 64 bit immediate values.

- All operations are non-blocking
- Return value indicates the status:
 - OK – operation is completed.
 - INPROGRESS – operation has started, but not completed yet.
 - NO_RESOURCE – cannot initiate the operation right now. The user might want to put this on a pending queue, or retry in a tight loop.
 - ERR_xx – other errors.
- Operations which may return INPROGRESS (get/atomics/zcopy) can get a completion handle.
 - User initializes the completion handle with a counter and a callback.
 - Each completion decrements the counter by 1, when it reaches 0 – the callback is called.

```
typedef ucs_status_t (*uct_am_callback_t)(void *arg, void *data, size_t length,
                                          void *desc);

typedef void (*uct_pack_callback_t)(void *dest, void *arg, size_t length);

typedef void (*uct_completion_callback_t)(uct_completion_t *self);

typedef struct uct_completion uct_completion_t;
struct uct_completion {
    uct_completion_callback_t func;
    int count;
};

typedef uintptr_t uct_rkey_t;
typedef void * uct_mem_h;

ucs_status_t uct_ep_put_short(uct_ep_h ep, const void *buffer, unsigned length,
                             uint64_t remote_addr, uct_rkey_t rkey);

ucs_status_t uct_ep_put_bcopy(uct_ep_h ep, uct_pack_callback_t pack_cb,
                              void *arg, size_t length, uint64_t remote_addr,
                              uct_rkey_t rkey);

ucs_status_t uct_ep_put_zcopy(uct_ep_h ep, const void *buffer, size_t length,
                              uct_mem_h memh, uint64_t remote_addr,
                              uct_rkey_t rkey, uct_completion_t *comp);

ucs_status_t uct_ep_am_short(uct_ep_h ep, uint8_t id, uint64_t header,
                             const void *payload, unsigned length);

ucs_status_t uct_ep_atomic_cswap64(uct_ep_h ep, uint64_t compare, uint64_t swap,
                                   uint64_t remote_addr, uct_rkey_t rkey,
                                   uint64_t *result, uct_completion_t *comp);
```

- **ucp_context_h**

A global context for the application. For example, hybrid MPI/SHMEM library may create one context for MPI, and another for SHMEM.

- **ucp_worker_h**

Communication resources and progress engine context. One possible usage is to create one worker per thread. Contains the `uct_iface_h`'s of all selected transports.

- **ucp_ep_h**

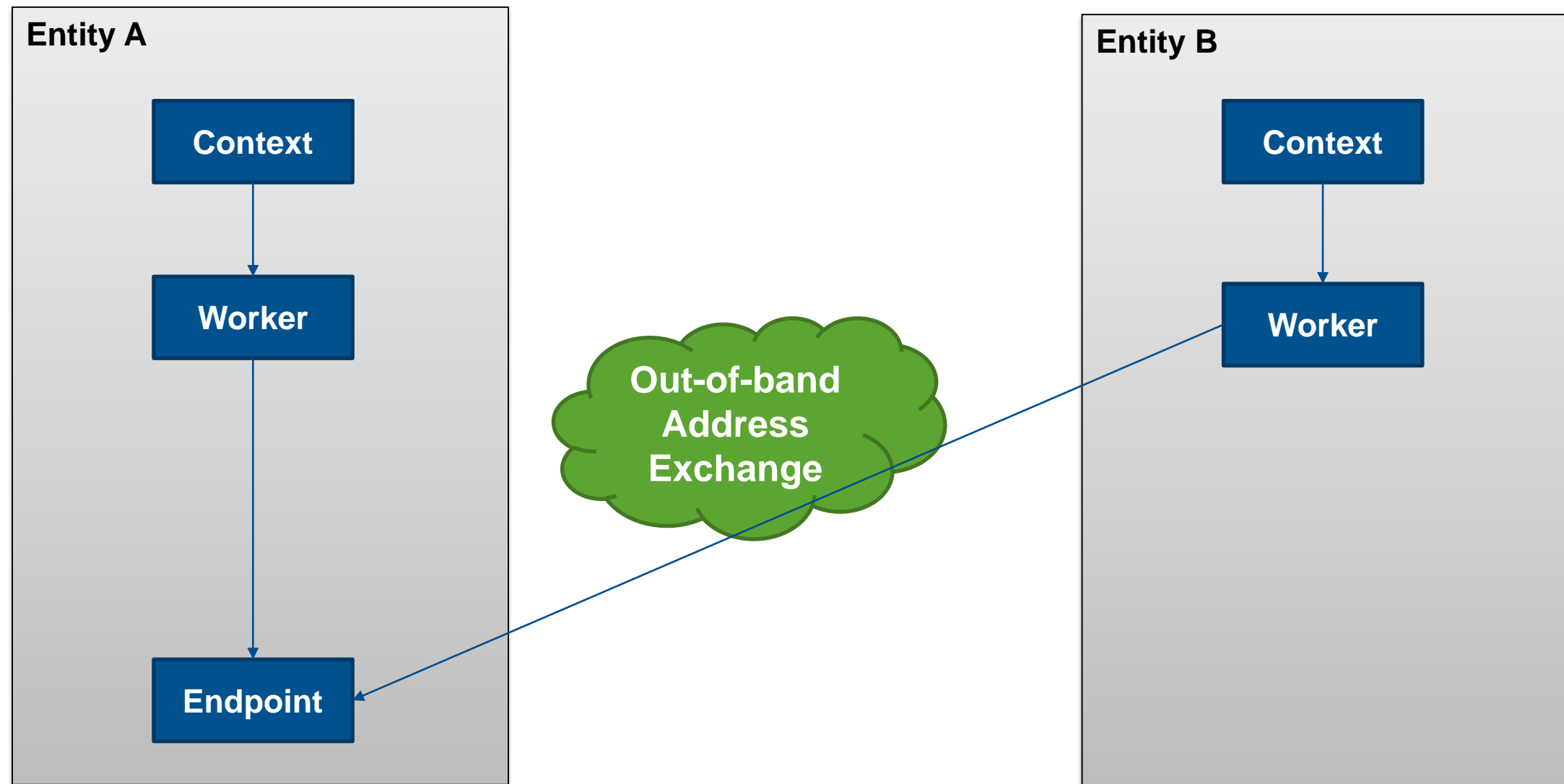
Connection to a remote worker. Used to initiate communications. Contains the `uct_ep_h`'s of currently active transports.

- **ucp_mem_h**

Handle to memory allocated or registered in the local process. Contains an array of `uct_mem_h`'s for currently active transports.

- **ucp_rkey_h**

Remote key handle, contains an array of `uct_rkey_t`'s.



- Tag-matched send/receive
 - Blocking / Non-blocking
 - Standard / Synchronous / Buffered
- Remote memory operations
 - Blocking put, get, atomics
 - Non-blocking – TBD
- Data is specified as buffer and length
 - No size limit
 - May register the buffer and use zero copy


```
typedef uint64_t ucp_tag_t;
```

```
ucs_status_t ucp_worker_create(ucp_context_h context, ucs_thread_mode_t thread_mode,  
                               ucp_worker_h *worker_p);
```

```
ucs_status_t ucp_worker_get_address(ucp_worker_h worker, ucp_address_t **address_p,  
                                    size_t *address_length_p);
```

```
ucs_status_t ucp_ep_create(ucp_worker_h worker, ucp_address_t *address,  
                           ucp_ep_h *ep_p);
```

```
ucs_status_t ucp_put(ucp_ep_h ep, const void *buffer, size_t length,  
                    uint64_t remote_addr, ucp_rkey_h rkey);
```

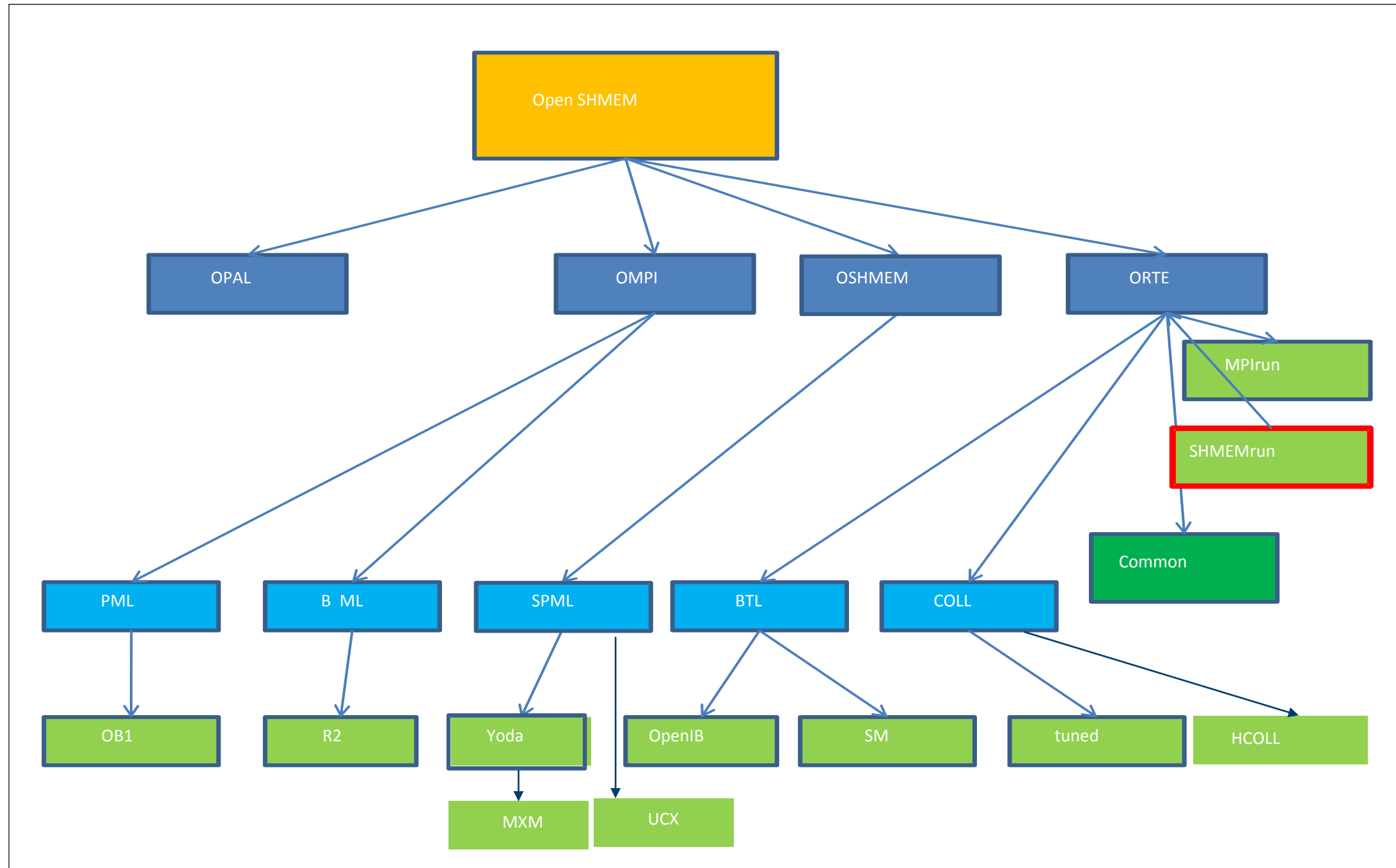
```
ucs_status_t ucp_get(ucp_ep_h ep, void *buffer, size_t length,  
                    uint64_t remote_addr, ucp_rkey_h rkey);
```

- Datatype engine
- Multi-threading
- HW tag matching
- UMR
- GPU support

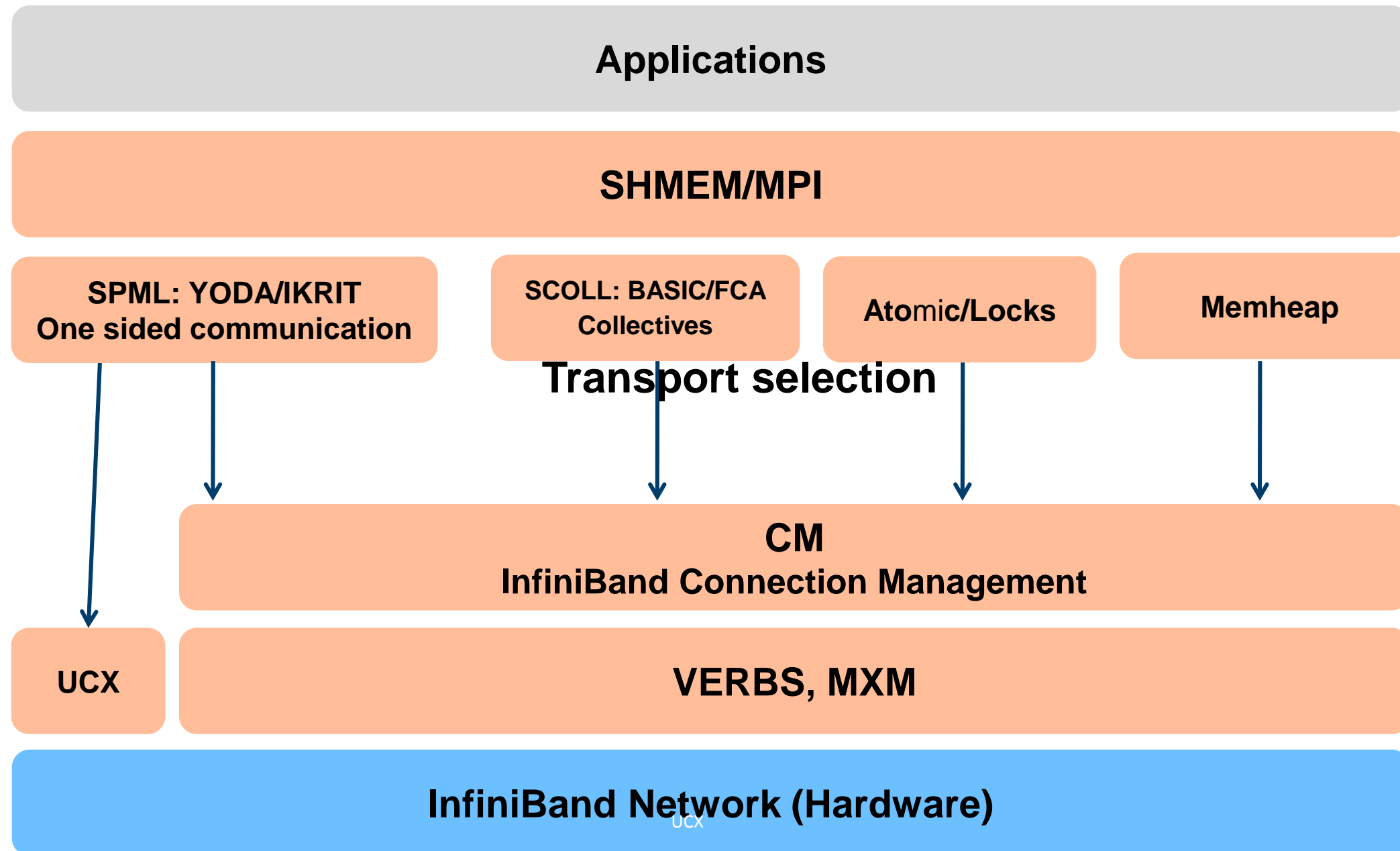
OpenSHMEM Support

- Strong interest and investment amongst National Labs and other Government agencies in OpenSHMEM.
- Currently, there exists little support for SHMEM over IB.
- One route to take: implement OpenSHMEM by leveraging existing and mature technologies (e.g. OMPI, MVAPICHX.)
- Open MPI is an one of the ideal platform given its MCA architecture and strong IB support in the BTL's as well as support for vendor provided libraries (MXM, FCA)
- OSHMEM 1.3 compliant in OMPI 2.1
- Goals:
 - Reuse existing OFA code and infrastructure
 - Expose hardware capabilities through libraries
 - Allow for integration of software libraries
 - MXM, UCX for p2p
 - FCA for collectives

SHMEM Integration with Open MPI



SHMEM Implementation Structure



Profiler

■ To profile MPI API:

- \$ export IPM_KEYFILE=\$HPCX_HOME_IPM_DIR/etc/ipm_key_mpi
- \$ export IPM_LOG=FULL
- \$ export LD_PRELOAD=\$HPCX_HOME_IPM_DIR/lib/libipm.so
- \$ mpirun -x LD_PRELOAD
- \$ \$HPCX_HOME_IPM_DIR/bin/ipm_parse -full outfile.xml
- \$ \$HPCX_HOME_IPM_DIR/bin/ipm_parse -html outfile.xml

```
##IPMv2.0.2#####
#
# command : /home/hpcg/hpcg-apex/bin/xhpcg
# start   : Thu Jan 14 15:17:53 2016 host    : thor031
# stop    : Thu Jan 14 15:21:08 2016 wallclock : 195.54
# mpi_tasks : 28 on 1 nodes %comm : 1.54
# mem [GB] : 28.10 gflop/sec : 0.00
#
#      : [total] <avg> min max
# wallclock : 5471.88 195.42 195.26 195.54
# MPI : 84.08 3.00 0.54 5.50
# %wall :
# MPI : 1.54 0.28 2.82
# #calls :
# MPI : 1569752 56062 40994 62090
# mem [GB] : 28.10 1.00 1.00 1.01
#
#      [time] [count] <%wall>
# MPI_Allreduce 38.32 15540 0.70
# MPI_Send 28.33 485208 0.52
# MPI_Wait 17.05 485208 0.31
# MPI_Irecv 0.34 485208 0.01
# MPI_Comm_size 0.03 49252 0.00
# MPI_Comm_rank 0.01 49252 0.00
# MPI_Bcast 0.00 28 0.00
# MPI_Init 0.00 28 0.00
# MPI_Finalize 0.00 28 0.00
#
#####
```

10643

Load Balance

Communication Balance

Message Buffer Sizes

Communication Topology

Switch Traffic

Memory Usage

Executable Info

Host List

Environment

Developer Info

powered by IPM

command: /home/hpcg/hpcg-apex/bin/xhpcg

codename: unknown

state: unknown

username: hpcg

group:

host: thor030 (x86_64_Linux)

mpi_tasks: 56 on 2 hosts

start: 01/14/16/15:17:52

wallclock: 1.96257e+02 sec

stop: 01/14/16/15:21:08

%comm: 1.76377148331015

total memory: 56.0554400000001 gbytes

total gflop/sec: 0

switch(send): 0 gbytes

switch(recv): 0 gbytes

Computation

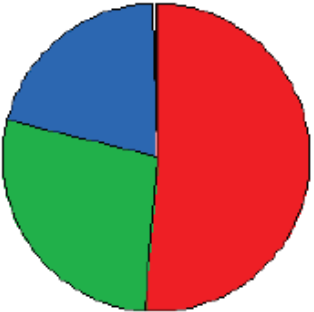
Event

Count

Pop

Communication

% of MPI Time



HPM Counter Statistics

Event	Ntasks	Avg	Min(rank)	Max(rank)
-------	--------	-----	-----------	-----------

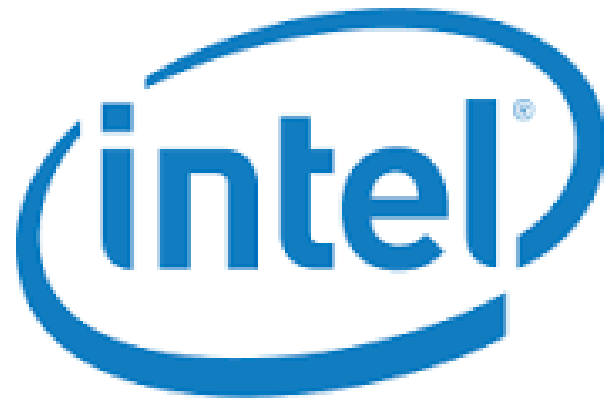
Communication Event Statistics (100.00% detail, 4.6909e-04 error)

	Buffer Size	Ncalls	Total Time	Avg Time	Min Time	Max Time	%MPI	%Wall
MPI_Allreduce	8	31080	99.143	3.190e-03	7.868e-06	4.875e-01	51.15	0.90
MPI_Send	81920	155524	32.542	2.092e-04	2.503e-05	2.210e-02	16.79	0.30
MPI_Send	20480	111156	19.657	1.768e-04	5.960e-06	1.445e-02	10.14	0.18
MPI_Wait	8	253152	13.871	5.479e-05	0.000e+00	2.908e-02	7.16	0.13
MPI_Wait	768	213516	8.445	3.955e-05	0.000e+00	1.954e-02	4.36	0.08
MPI_Wait	384	152604	6.225	4.079e-05	0.000e+00	8.339e-03	3.21	0.06
MPI_Wait	192	152604	3.710	2.431e-05	0.000e+00	5.187e-03	1.91	0.03
MPI_Wait	81920	155524	3.201	2.058e-05	0.000e+00	1.165e-02	1.65	0.03
MPI_Wait	20480	111156	1.771	1.593e-05	0.000e+00	8.301e-03	0.91	0.02
MPI_Wait	5120	111156	1.695	1.525e-05	0.000e+00	4.824e-03	0.87	0.02

- To troubleshoot for MPI load imbalance
 - Apply blocking before MPI collective operations
 - Show the effect when processes not synchronized before entering into MPI collective ops
- Instrumentation can be applied on per-collective basis
 - `$ export IPM_ADD_BARRIER_TO_REDUCE=1`
 - `$ export IPM_ADD_BARRIER_TO_ALLREDUCE=1`
 - `$ export IPM_ADD_BARRIER_TO_GATHER=1`
 - `$ export IPM_ADD_BARRIER_TO_ALL_GATHER=1`
 - `$ export IPM_ADD_BARRIER_TO_ALLTOALL=1`
 - `$ export IPM_ADD_BARRIER_TO_ALLTOALLV=1`
 - `$ export IPM_ADD_BARRIER_TO_BROADCAST=1`
 - `$ export IPM_ADD_BARRIER_TO_SCATTER=1`
 - `$ export IPM_ADD_BARRIER_TO_SCATTERV=1`
 - `$ export IPM_ADD_BARRIER_TO_GATHERV=1`
 - `$ export IPM_ADD_BARRIER_TO_ALLGATHERV=1`
 - `$ export IPM_ADD_BARRIER_TO_REDUCE_SCATTER=1`

PMix

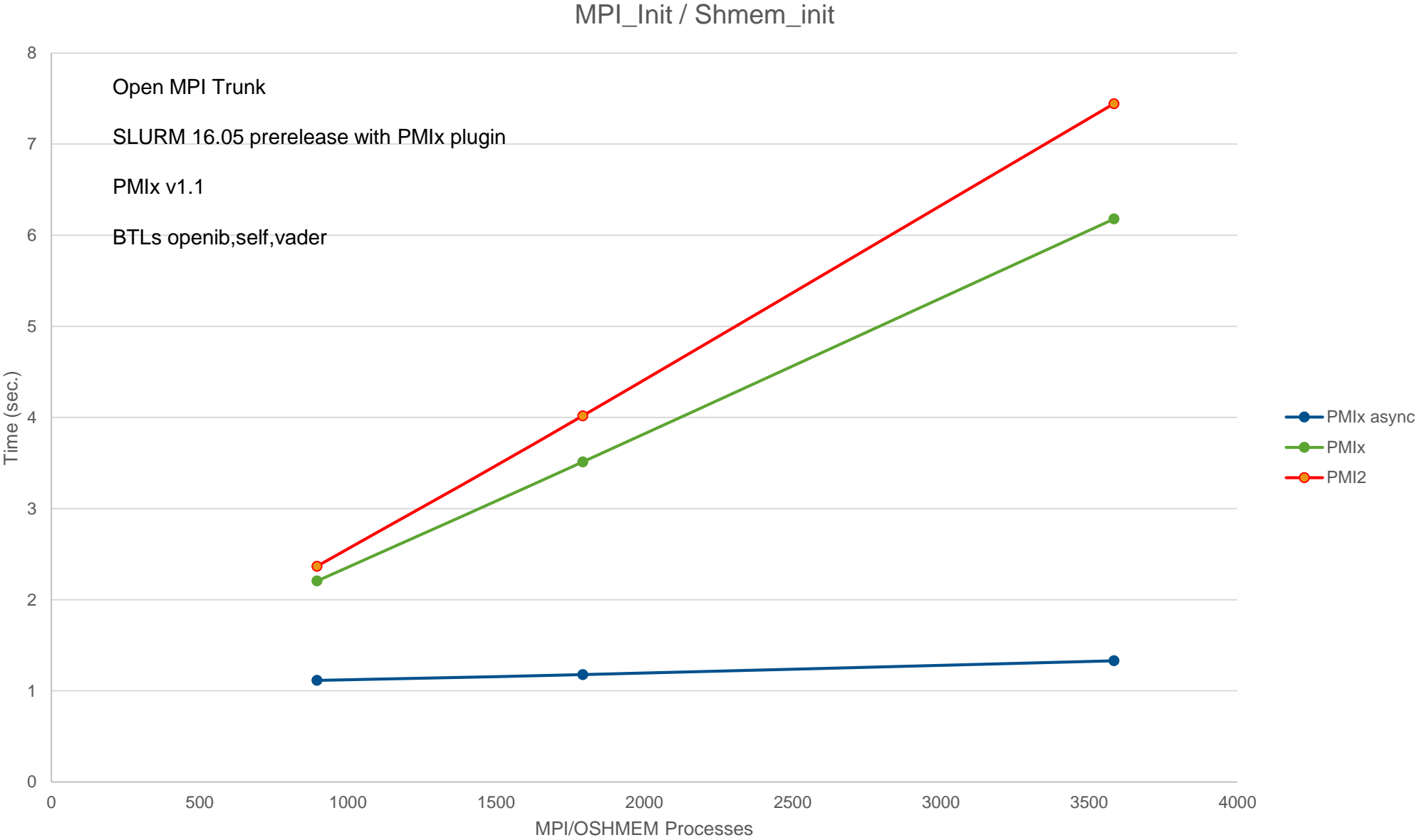
- Collaborative open source effort led by Intel, Mellanox Technologies, IBM, Adaptive Computing, and SchedMD...



- Exascale launch times are a hot topic
 - Desire: reduce from many minutes to few seconds
 - Target: $O(10^6)$ MPI processes on $O(10^5)$ nodes thru MPI_Init in < 30 seconds
- New programming models are exploding
 - Driven by need to efficiently exploit scale vs. resource constraints
 - Characterized by increased app-RM integration

- Worked closely with customers, OEMs, and open source community to design a scalable API that addresses measured limitations of PMI2
 - Data driven design.
- Led to the PMIx v1.0 API
- Implementation and imminent release of PMIx v1.1
- Significant architectural changes in Open MPI to support direct modex
 - “Add procs” in bulk MPI_Init → “Add proc” on-demand on first use outside MPI_init.
 - Available in the OMPI v2.x release.
- Integrated PMIx into Open MPI v2.x
 - For native launching as well as direct launching under supported RMs.
 - For mpirun launched jobs, ORTE implements PMIx callbacks.
 - For srun launched jobs, SLURM implements PMIx callbacks in the PMIx plugin.
 - Client side framework added to OPAL with components for
 - Cray PMI
 - PMI1
 - PMI2
 - PMIx
 - backwards compatibility with PMI1 and PMI2.
- Implemented and submitted upstream SLURM PMIx plugin
 - released in SLURM 16.05
- PMIx unit tests integrated into Jenkins test harness

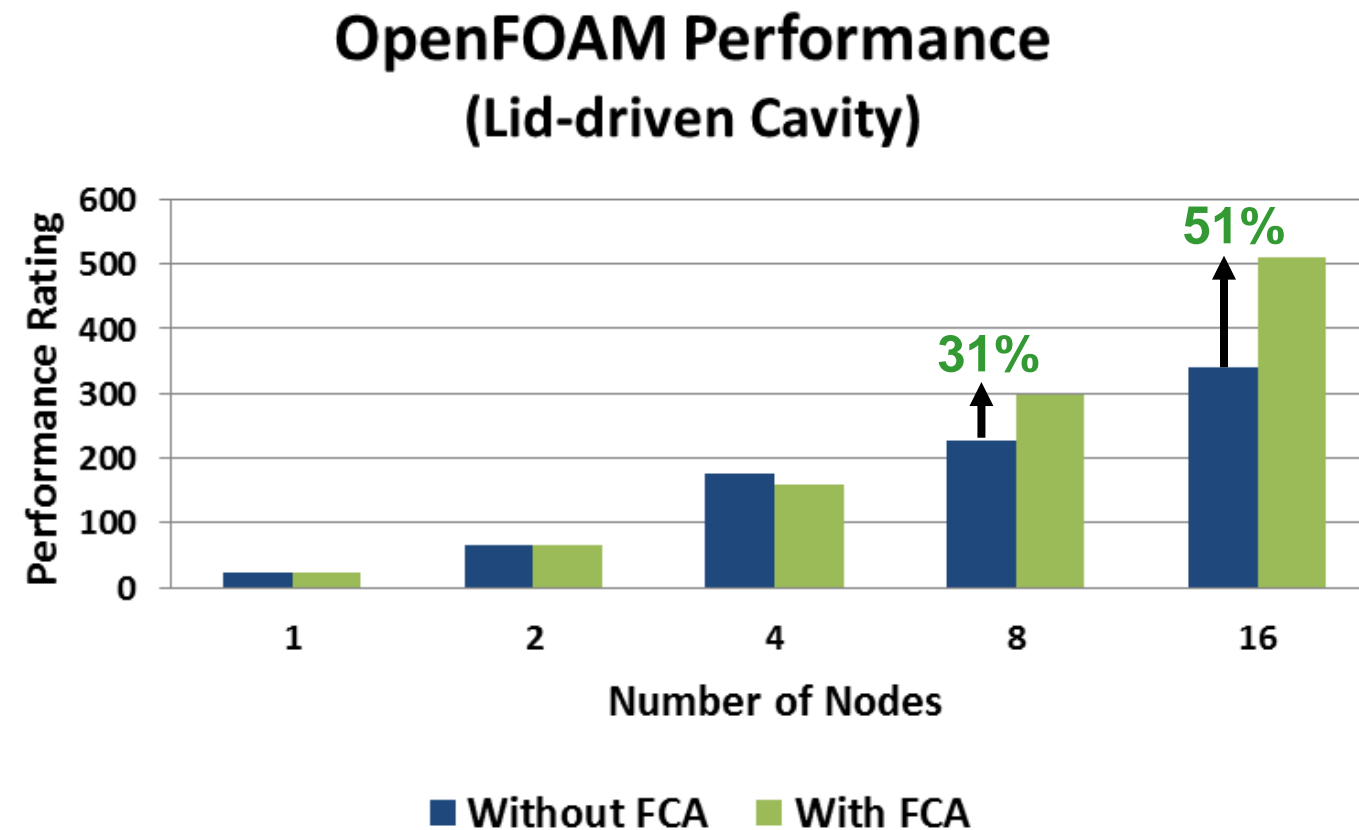
```
srun --mpi=xxx hello_world
```



- API is implemented and performing well in a variety of settings
 - Server integrated in OMPI for native launching and in SLURM as PMIx plugin for direct launching.
- PMIx shows improvement over other state-of-the-art PMI2 implementations when doing a full modex
 - Data blobs versus encoded metakeys
 - Data scoping to reduce the modex size (local, remote, global)
 - Persistence for data.
 - Shared memory for in node copy
- PMIx supported direct modex significantly outperforms full modex operations for BTL/MTLs that can support this feature
- Direct modex still scales as $O(N)$
- Efforts and energy should be focused on daemon bootstrap problem
- Instant-on capabilities could be used to further reduce daemon bootstrap time

Application Case Studies with HPCX

- FCA enables nearly 51% performance gain at 16 nodes / 256 cores
 - Bigger advantage expected at higher node count / core count

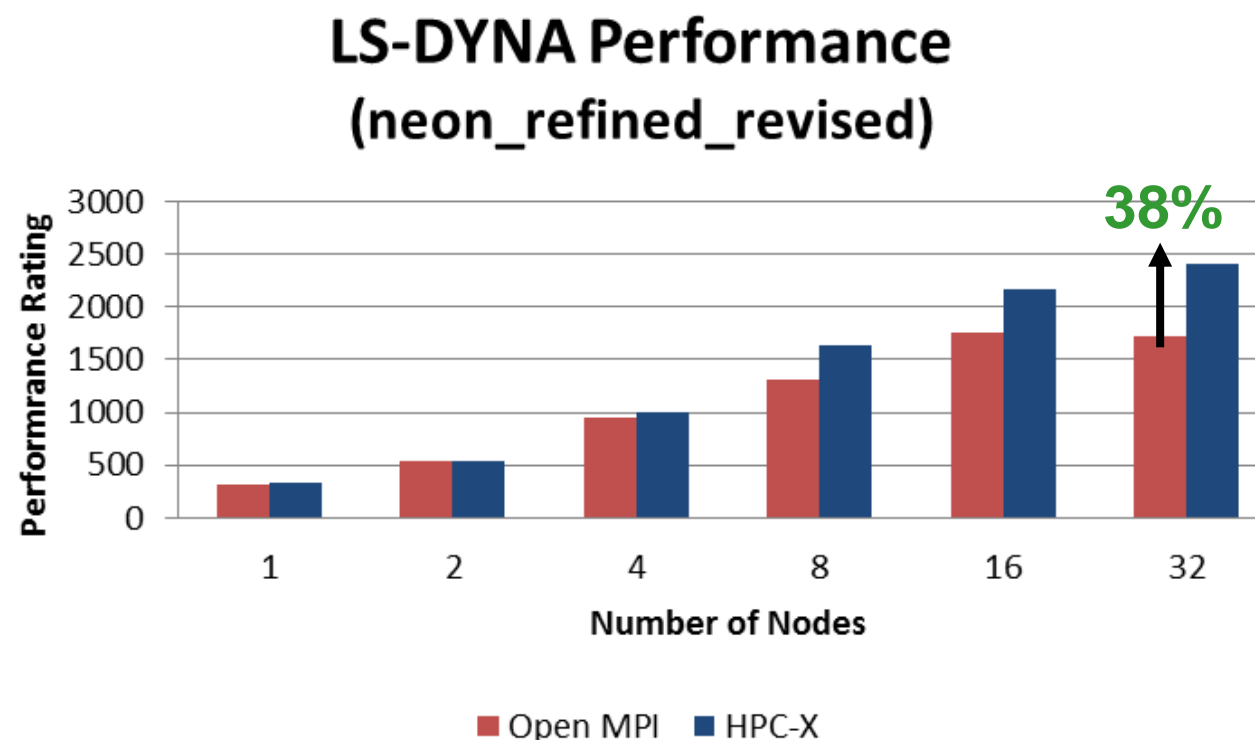


LS-DYNA Performance – MPI Optimization



- FCA and MXM enhance LS-DYNA performance at scale for HPC-X
 - Open MPI and HPC-X are based on the Open MPI distribution
 - The “yalla” PML, UD transport and memory optimization in HPC-X reduce overhead
 - MXM provides a speedup of 38% over un-tuned baseline run at 32 nodes (768 cores)
- MCA parameters for MXM:
 - For enabling MXM:

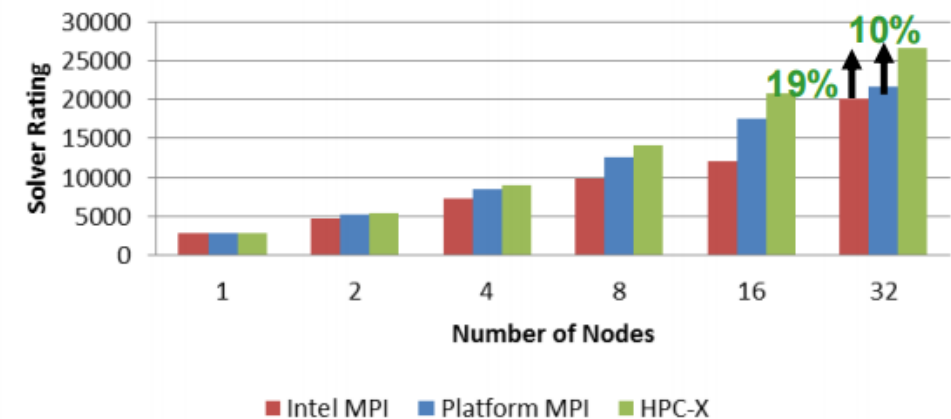
```
-mca btl_sm_use_knem 1 -mca pml yalla -x MXM_TLS=ud,shm,self -x MXM_SHM_RNDV_THRESH=32768 -x MXM_RDMA_PORTS=mlx5_0:1
```



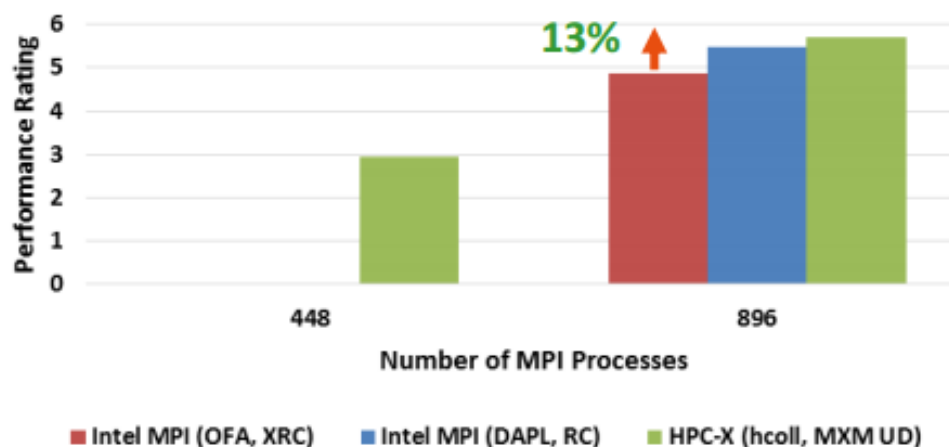
Mellanox Delivers Highest Applications Performance (HPC-X)



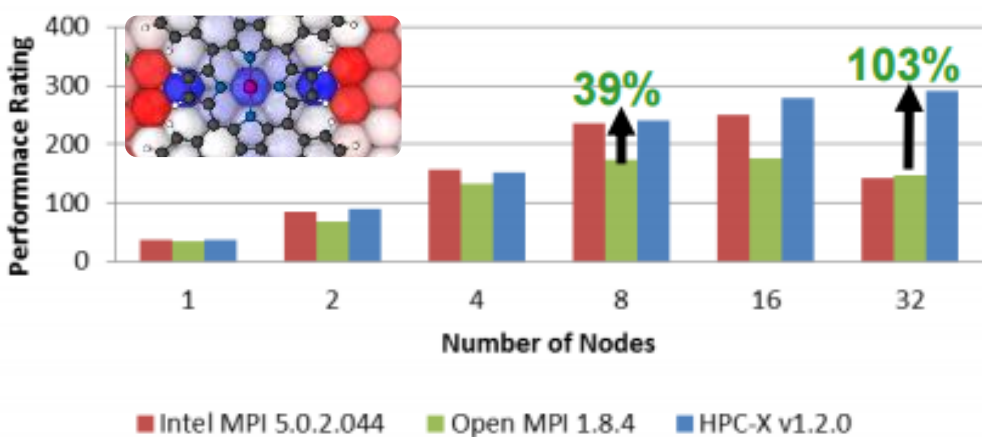
ANSYS Fluent 15.0.7 Performance
(eddy_417k)



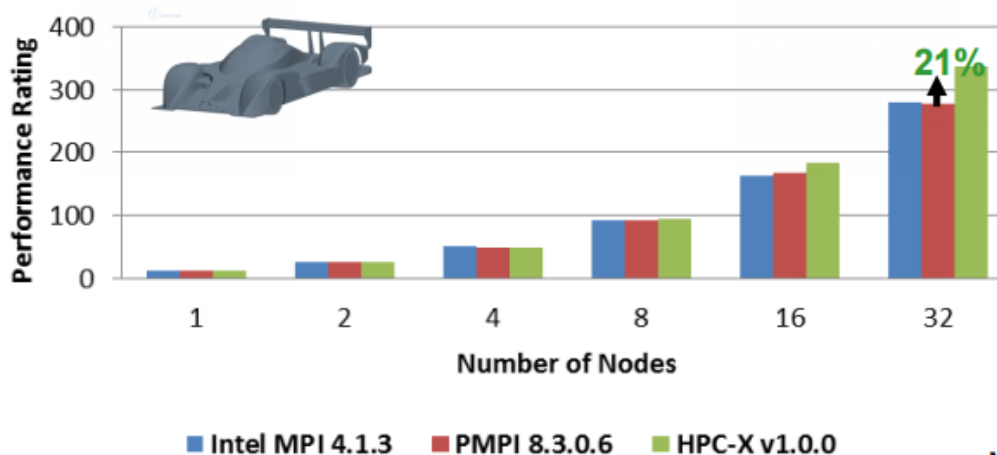
OpenFOAM Performance
(Automotive Benchmark, pimpleFoam)



Quantum ESPRESSO Performance
(AUSURF112)

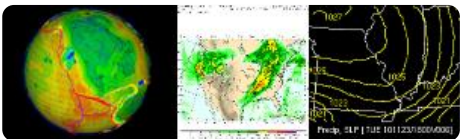
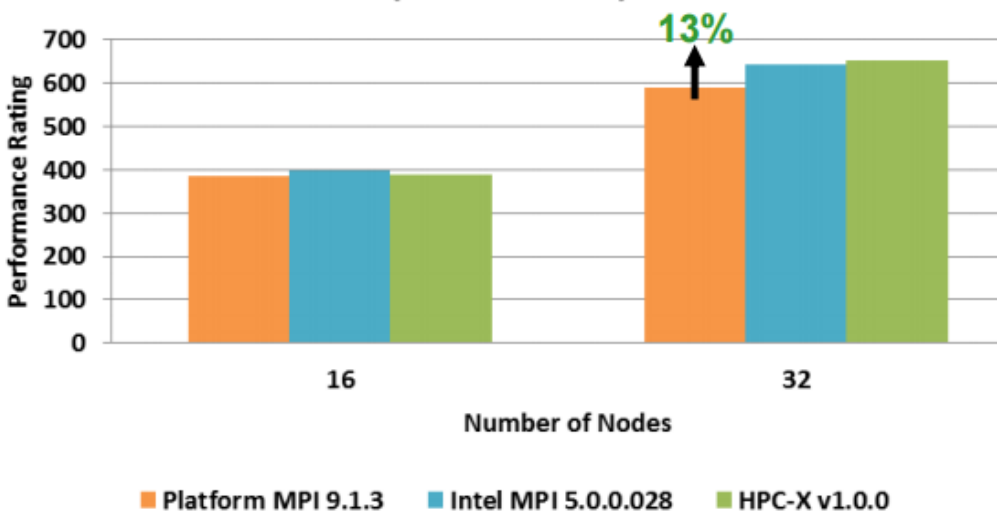


STAR-CCM+ Performance
(lemans_poly_17m)



Enabling Highest Applications Scalability and Performance

WRF Performance
(conus12km)



■ ANSYS Fluent

- HPC-X provides 19% higher performance than Intel MPI on 32 nodes
- http://www.hpcadvisorycouncil.com/pdf/Fluent_Analysis_and_Profiling_Intel_E5_2680v2.pdf

■ CD-adapco STAR-CCM+

- Up to 21% higher than Intel MPI and Platform MPI, at 32 nodes
- http://www.hpcadvisorycouncil.com/pdf/STAR-CCM_Analysis_Intel_E5_2680_V2.pdf

■ OpenFOAM

- HPC-X delivers 13% gain over Intel MPI on 32 nodes
- HPC-X demonstrates 24% improvement over Open MPI
- http://www.hpcadvisorycouncil.com/pdf/OpenFOAMConf2015_PakLui.pdf

■ QuantumESPRESSO

- HPC-X delivers 103% gain over Intel MPI on 32 nodes
- http://www.hpcadvisorycouncil.com/pdf/QuantumEspresso_Performance_Analysis_Intel_Haswell.pdf

■ WRF

- HPC-X delivers 13% performance improvement over Platform MPI on 32 nodes
- http://www.hpcadvisorycouncil.com/pdf/WRF_Analysis_and_Profiling_Intel_E5-2680v2.pdf

- Mellanox solutions provide a proven, scalable and high performance end-to-end connectivity
- Flexible, support all compute architectures: x86, ARM, GPU, FPGA etc.
- Standards-based (InfiniBand, Ethernet), supported by large eco-system
- Higher performance: 100Gb/s, 0.7usec latency, 150 million messages/sec
- HPC-X software provides leading performance for MPI, OpenSHMEM/PGAS and UPC
- Superior applications offloads: RDMA, Collectives, scalable transport
- Backward and future compatible



Thank You