ENABLER OF CO-DESIGN

UCF

Unified Communication X (UCX)

Rich Graham

March 2019

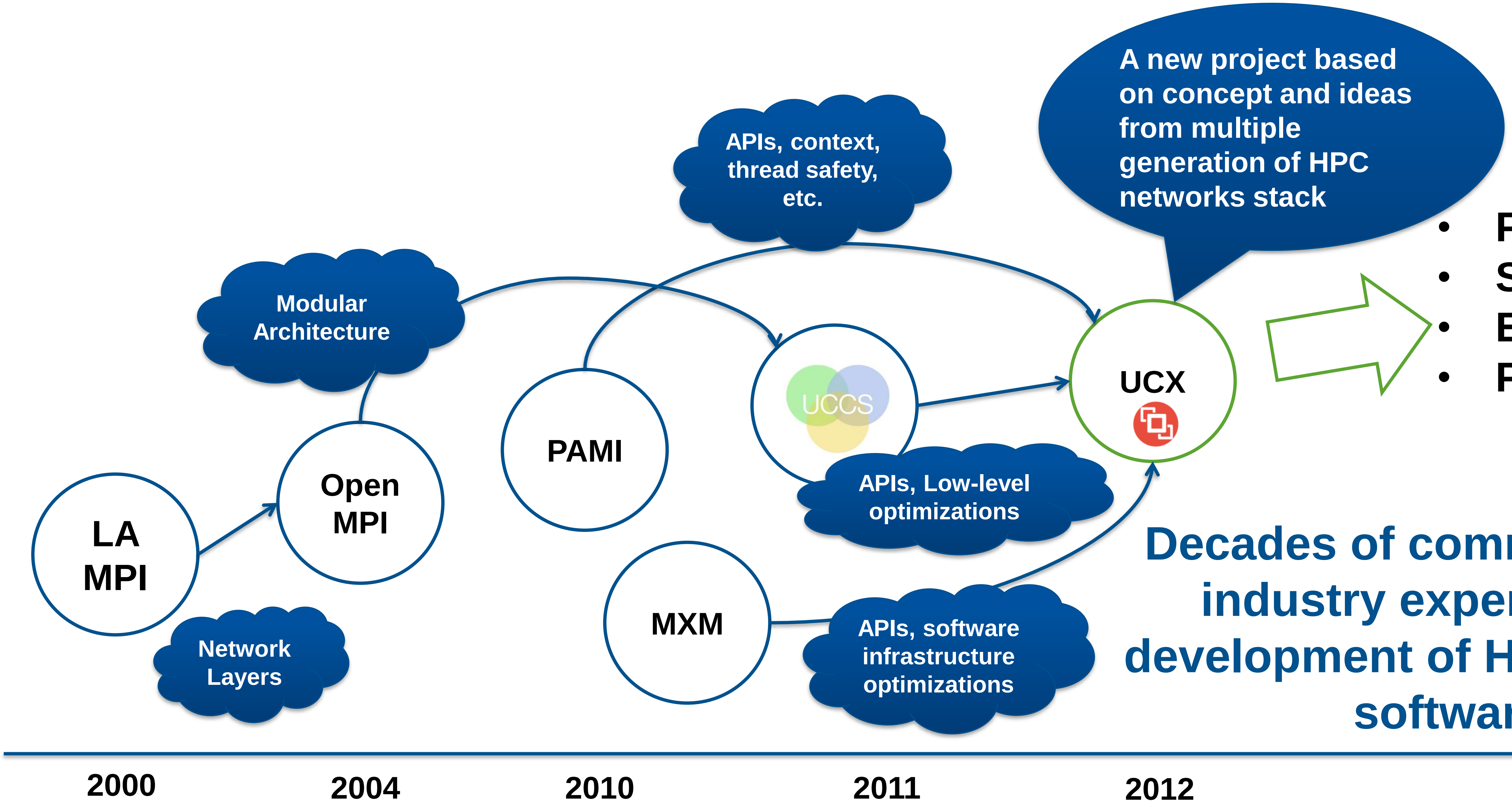
UCF Consortium

- Mission:
  - Collaboration between industry, laboratories, and academia to create production grade communication frameworks and open standards for data centric and high-performance applications

- Projects
  - UCX – Unified Communication X
  - Open RDMA

- Board members
  - **Jeff Kuehn**, UCF Chairman (Los Alamos National Laboratory)
  - **Gilad Shainer**, UCF President (Mellanox Technologies)
  - **Pavel Shamis**, UCF treasurer (Arm)
  - **Brad Benton**, Board Member (AMD)
  - **Duncan Poole**, Board Member (NVIDIA)
  - **Pavan Balaji**, Board Member (Argonne National Laboratory)
  - **Sameh Sharkawi**, Board Member (IBM)
  - **Dhabaleswar K. (DK) Panda**, Board Member (Ohio State University)
  - **Steve Poole**, Board Member (Open Source Software Solutions)

UCX - History



- Performance
- Scalability
- Efficiency
- Portability

Decades of community and industry experience in development of HPC network software

UCX Framework Mission

- Collaboration between industry, laboratories, government (DoD, DoE), and academia
- Create open-source production grade communication framework for HPC applications
- Enable the highest performance through co-design of software-hardware interfaces

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

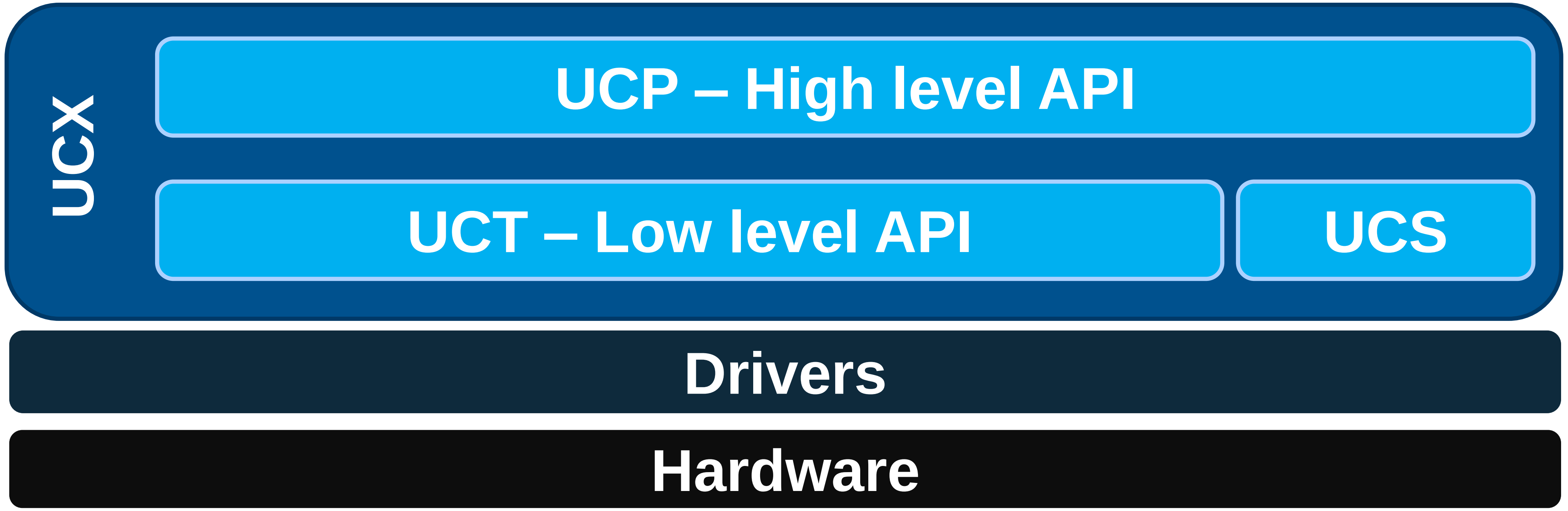
Support for Infiniband, Cray, various shared memory (x86-64, Power, ARMv8), GPUs

Co-design of Exascale Network APIs

UCX Framework

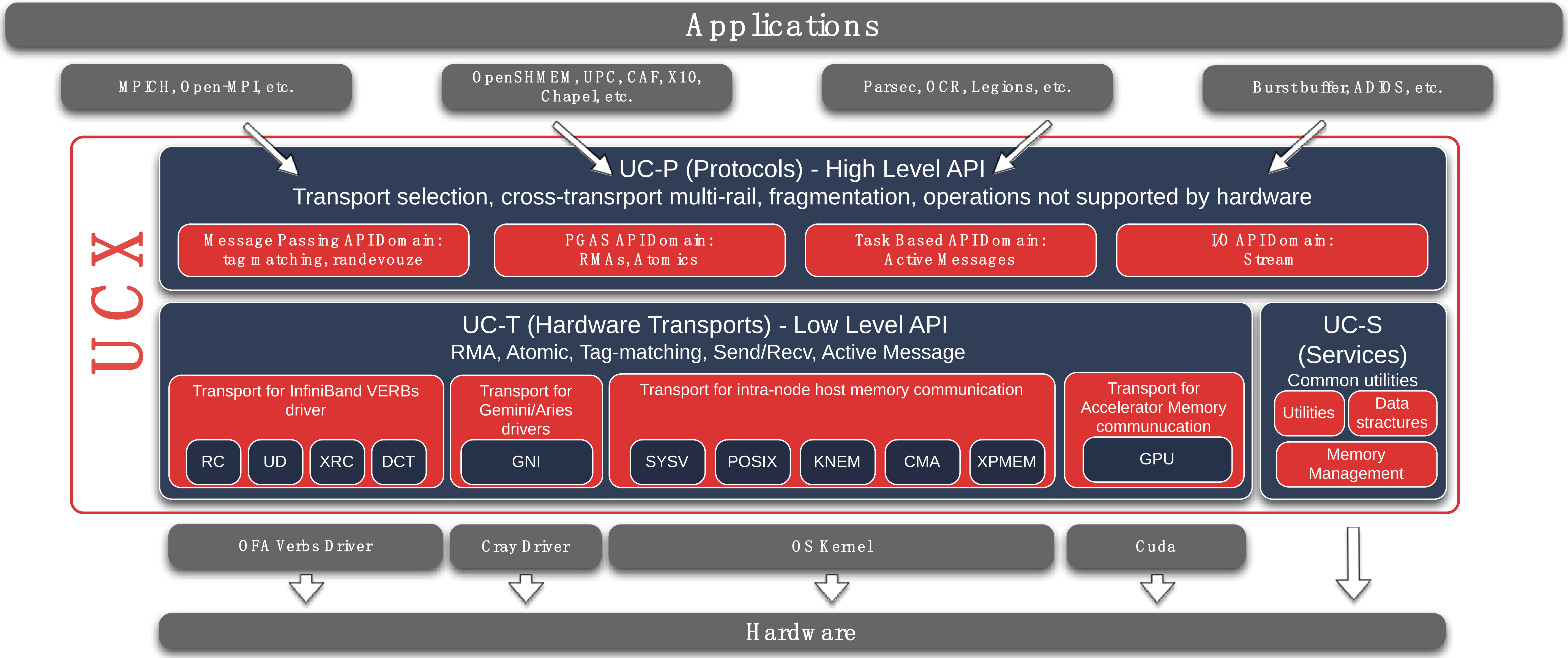
- UCX is a framework for network APIs and stacks
- UCX aims to unify the different network APIs, protocols and implementations into a single framework that is portable, efficient and functional
- UCX doesn’t focus on supporting a single programming model, instead it provides APIs and protocols that can be used to tailor the functionalities of a particular programming model efficiently
- When different programming paradigms and applications use UCX to implement their functionality, it increases their portability. As just implementing a small set of UCX APIs on top of a new hardware ensures that these applications can run seamlessly without having to implement it themselves

UCX Architecture



- **UCX framework** is composed of **three main components**.
- **UCP** layer is the protocol layer and supports all the functionalities exposed by the high-level APIs, meaning it emulates the features that are not implemented in the underlying hardware
- **UCT** layer is the transport layer that aims to provides a very efficient and low overhead access to the hardware resources
- **UCS** is a service layer that provides common data structures, memory management tools and other utilities

UCX High-level Overview



UCX Release 1.3.1

- v1.3.1
  - **Multi-rail** support for eager and rendezvous protocols
  - Added **stream-based communication** API
  - Added support for **GPU** platforms: Nvidia CUDA and AMD ROCM software stacks
  - Added API for **Client-Server** based connection establishment
  - Added support for **TCP** transport (Send/Receive semantics)
  - Support for InfiniBand **hardware tag-matching** for DC and accelerated transports
  - Added support for **tag-matching communications with CUDA** buffers
  - Initial support for **Java bindings**
  - **Progress engine** optimizations
  - Improved scalability of **software tag-matching** by using a hash table
  - Added transparent **huge-pages** allocator
  - Added **non-blocking flush and disconnect** semantics
  - Added registration cache for **KNEM**
  - Support fixed-address memory allocation via ucp_mem_map()
  - Added ucp_tag_send_nbr() API to avoid send request allocation
  - Support global addressing in all IB transports
  - Add support for external epoll fd and edge-triggered events
  - Added ucp_rkey_ptr() to obtain pointer for shared memory region

UCX Release 1.4

- V1.4.0
  - Support for installation with latest **AMD ROCm**
  - Support for latest **RDMA-CORE**
  - Support for **NVIDIA CUDA IPC** for intra-node GPU
  - Support for **NVIDIA CUDA** memory allocation cache for mem-type detection
  - Support for latest **Mellanox devices (200Gb/s)**
  - Support for **NVIDIA GPU managed memory**
  - Support for **bitwise** (OpenSHMEM v1.4) atomics operations

X86, Power8/9, arm

State-of-the-art support for GP-GPU

InfiniBand, RoCEv1/v2, Gemini/Aries, Shared Memory, TCP/Ethernet (Beta)

UCX Release 1.5

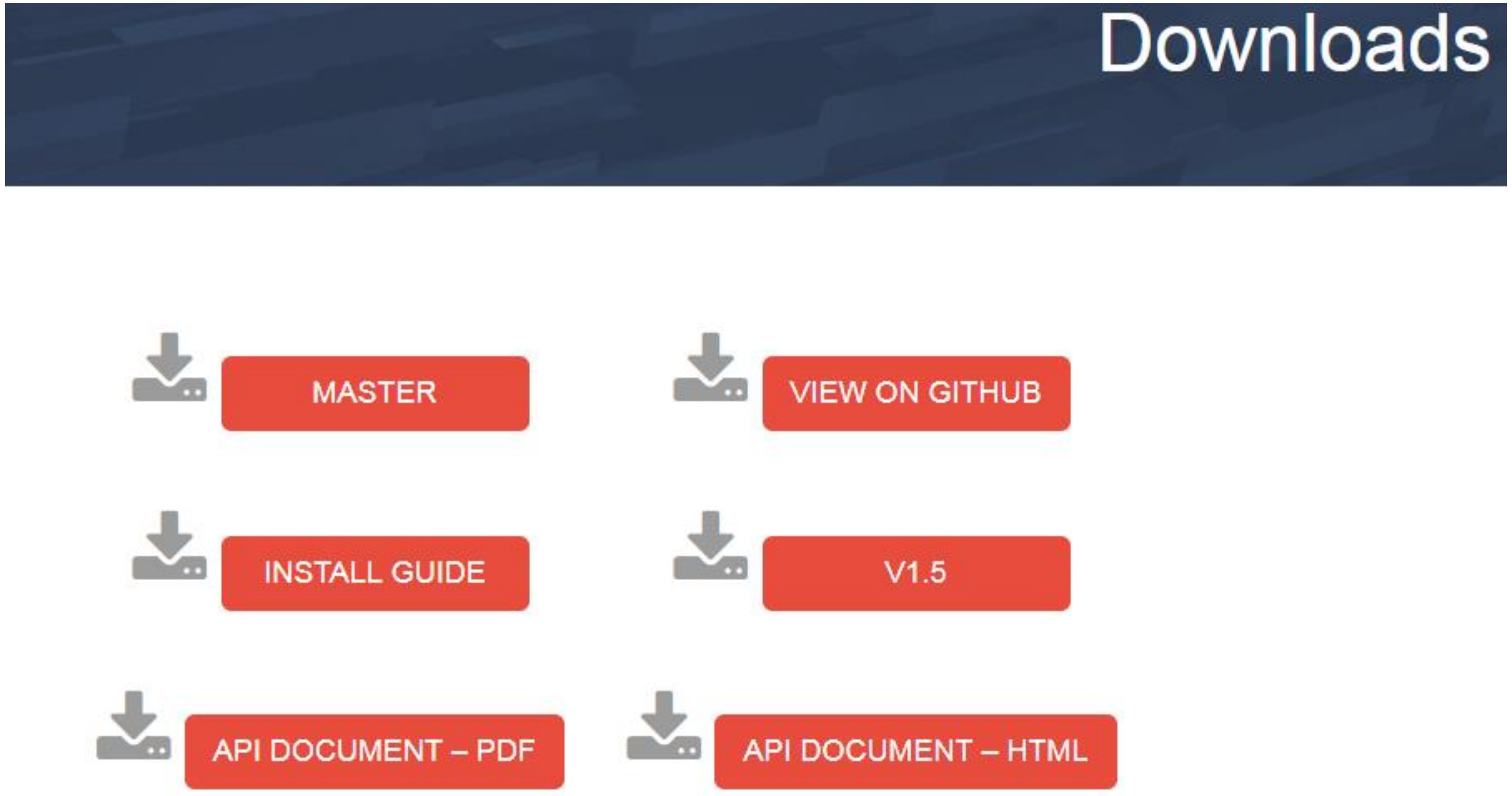
- V1.5.0
  - New **emulation** mode enabling comprehensive UCX functionality (Atomic, Put, Get, etc) over TCP and legacy interconnects that don't implement full RDMA semantics.
  - New **non-blocking API** for all one-sided operations.
  - New **client/server connection establishment** API
  - New advanced **statistic** capabilities (tag matching queues)

UCX Roadmap

- v1.6
  - Bugfixes and optimizations
  - IWARP
  - Active Message API
- v2.0
  - Updated API – not backward compatible with 1.x
    - Cleanup (remove deprecated APIs)
    - UCP request object redesign – improves future backward compatibility
  - Binary distribution will provide v1.x version of the library (in addition for 2.x) for backward compatibility
    - All codes should work as it is

Downloads

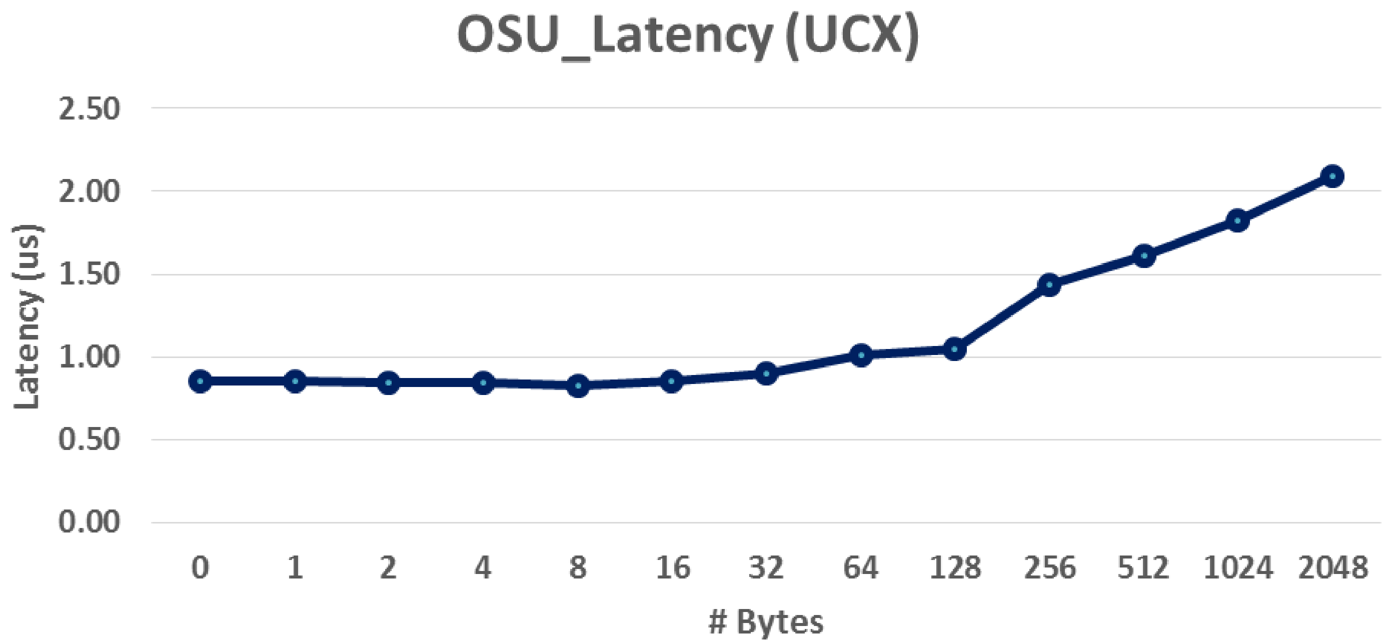
- http://www.openucx.org/downloads/

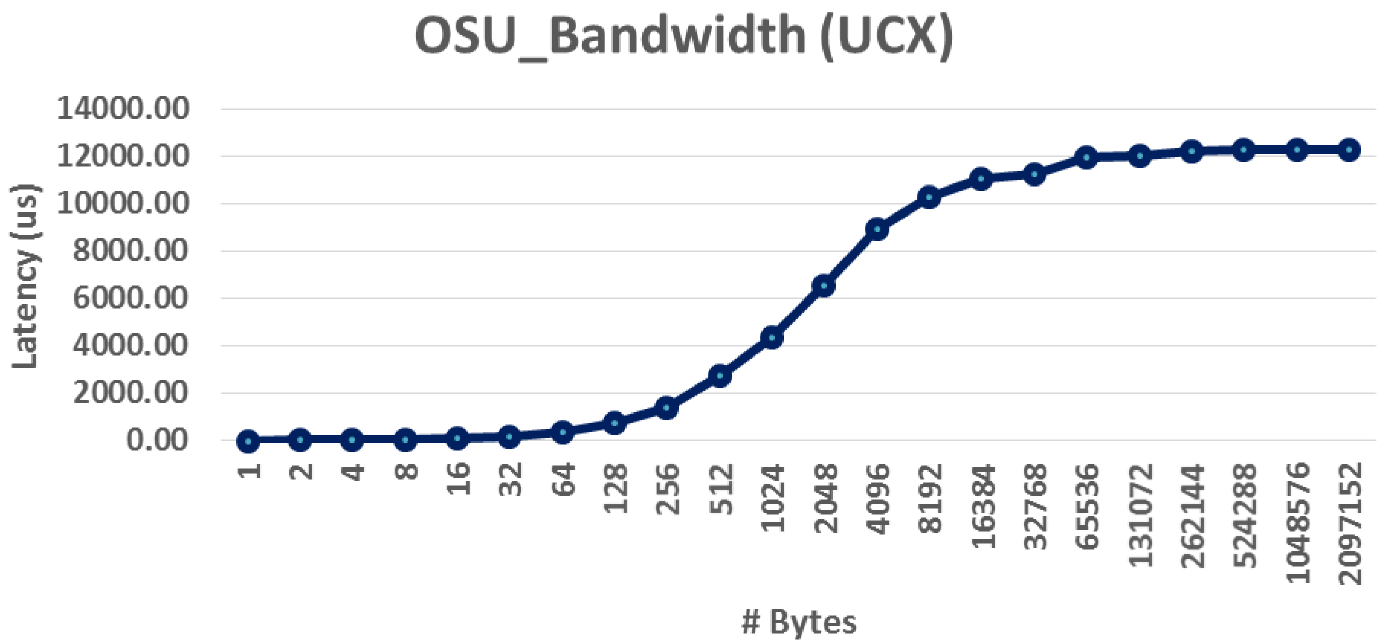


Integrations

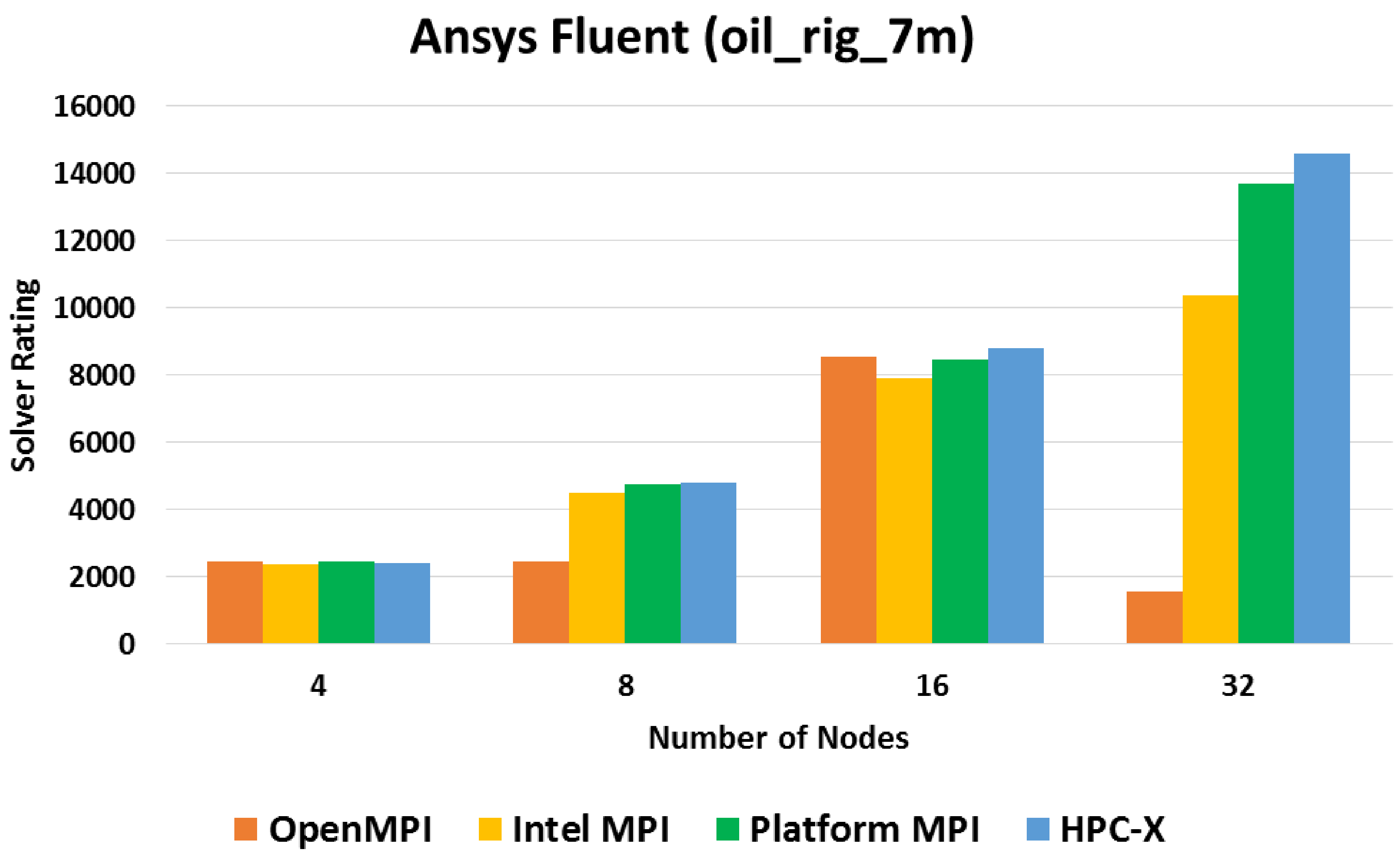
- Open MPI and OSHMEM
  - UCX replaces OpenIB BTL as default transport for InfiniBand and RoCE
  - New UCX BTL (by LANL)
- MPICH MPI
  - CH4 UCX
- OSSS SHMEM by StonyBrook and LANL
- Open SHMEM-X by ORNL
- Parsec (UTK)
- Intel Libfabrics/OFI
  - Powered by UCX !
- 3rd party commercial projects

Performance – MPI Latency and Bandwidth





Performance - Fluent



API Overview

UCP (Protocol Layer)

- Combine transports, devices, and operations, for optimal performance
  - Query transport capabilities and performance estimations
  - Select devices/transports according to reachability
  - Select best protocols for each data transfer type
  - For example: eager vs. rendezvous, bcopy vs. zcopy
- Unified transport infrastructure
  - Fragmentation (transport has limited MTU)
  - Emulate unsupported operations
  - Expose one-sided connection establishment
- Fill-in for missing hardware support
  - Software tag matching
  - Active-message based remote memory access (put/get) and atomics
- Multi-rail - Aggregate multiple devices to single logical connection

UCP Contexts

- ucp_context_h

  Top-level context for the application.
- ucp_worker_h

  Communication resources and progress engine context. A possible usage is to create one worker per thread or per CPU core.
- ucp_listener_h

  Listens for incoming connection requests on a specific port.
- ucp_ep_h

  Connection to a remote worker, used to send data to remote peer. Contains handles for all transport-level connections in use to the remote peer.
- 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 memory handle. Allows access to remote memory for one-sided operations and atomics.

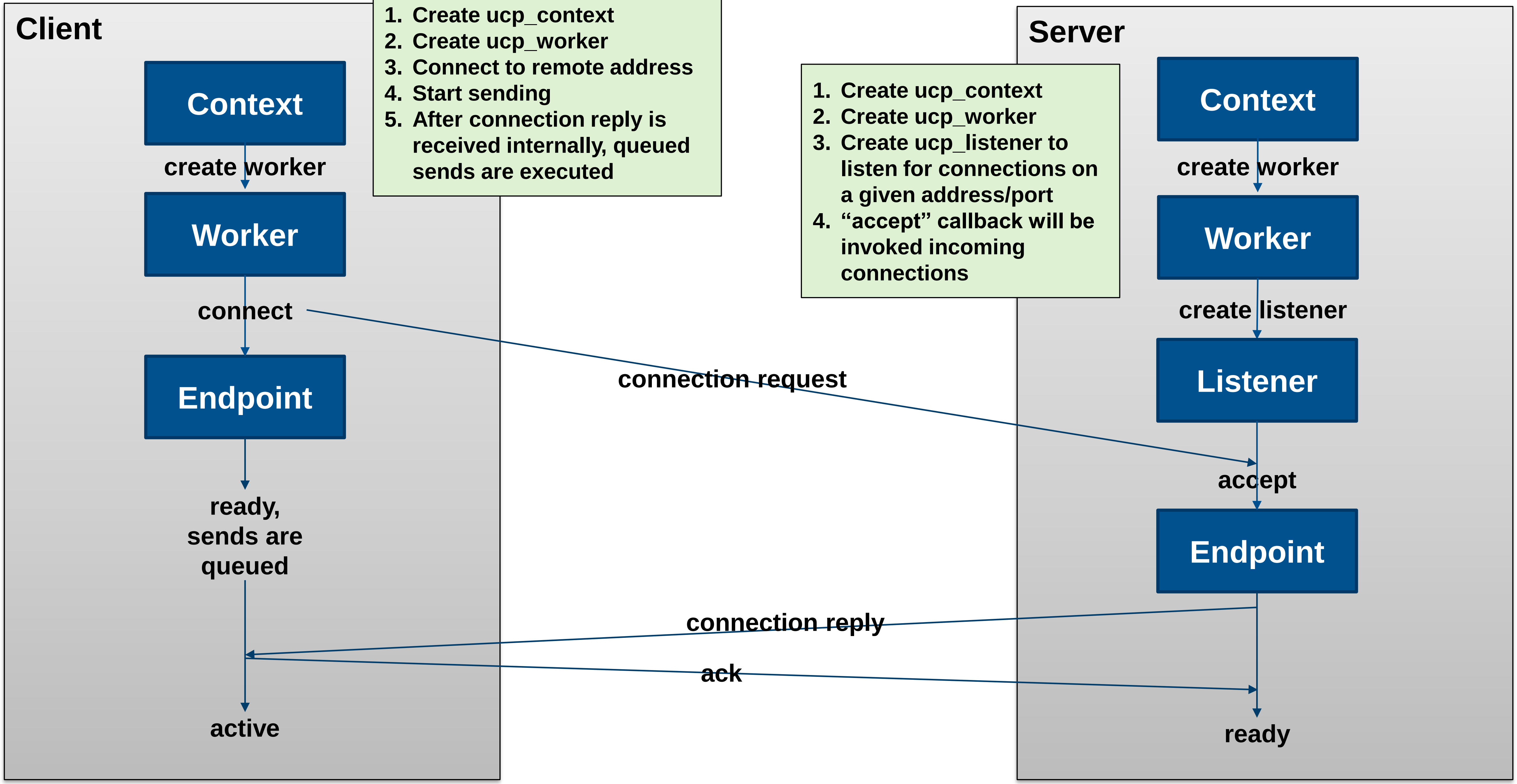
UCP Connection Establishment – Client/Server Model

Client

1. Create ucp_context
2. Create ucp_worker
3. Connect to remote address
4. Start sending
5. After connection reply is received internally, queued sends are executed

Server

1. Create ucp_context
2. Create ucp_worker
3. Create ucp_listener to listen for connections on a given address/port
4. "accept" callback will be invoked incoming connections



UCP API – Stream-Based Send/Receive

Use cases:

- Drop-in replacement for TCP sockets use cases (cloud, storage, …)
- In-place processing of streaming data

- Non-blocking send:

```
ucs_status_ptr_t ucp_stream_send_nb(ucp_ep_h ep, const void *buffer, size_t count,
                                    ucp_datatype_t datatype, ucp_send_callback_t cb,
                                    unsigned flags)
```

- Non-blocking receive:

```
ucs_status_ptr_t ucp_stream_recv_nb(ucp_ep_h ep, void *buffer,
                                    size_t count, ucp_datatype_t datatype,
                                    ucp_stream_recv_callback_t cb,
                                    size_t *length, unsigned flags)
```

- Fetch next data fragment from stream:

```
ucs_status_ptr_t ucp_stream_recv_data_nb(ucp_ep_h ep, size_t *length)
```

UCP API – Tag-Based Send/Receive

Use cases:

- Implementing MPI-1 standard (OpenMPI, MPICH)

- Non-blocking send:

```
ucs_status_ptr_t ucp_tag_send_nb(ucp_ep_h ep, const void *buffer,
                                 size_t count, ucp_datatype_t datatype,
                                 ucp_tag_t tag, ucp_send_callback_t cb)
```

- Non-blocking receive:

```
ucs_status_ptr_t ucp_tag_recv_nb(ucp_worker_h worker, void *buffer,
                                 size_t count, ucp_datatype_t datatype,
                                 ucp_tag_t tag, ucp_tag_t tag_mask,
                                 ucp_tag_recv_callback_t cb)
```

UCP API – Remote Memory Access

Use cases:

- Implementing MPI-RMA operations (OpenMPI, MPICH)
- Implementing PGAS/SHMEM stack (OpenSHMEM)
- Bulk data transfer after synchronizing with control messages

Write to remote memory:

```
ucs_status_ptr_t ucp_put_nb(ucp_ep_h ep, const void *buffer, size_t length,
                            uint64_t remote_addr, ucp_rkey_h rkey,
                            ucp_send_callback_t cb)
```

- Implicit non-blocking variant: **ucp_put_nbi**

Read from remote memory:

```
ucs_status_ptr_t ucp_get_nb(ucp_ep_h ep, void *buffer, size_t length,
                            uint64_t remote_addr, ucp_rkey_h rkey,
                            ucp_send_callback_t cb)
```

- Implicit non-blocking variant: **ucp_get_nbi**

UCP API – Atomic Operations

Use cases:

- MPI-RMA
- PGAS/SHMEM

- Perform atomic operation on remote memory:

```
ucs_status_t ucp_atomic_fetch_nb(ucp_ep_h ep, ucp_atomic_fetch_op_t opcode,
                                 uint64_t value, void *result, size_t op_size,
                                 uint64_t remote_addr, ucp_rkey_h rkey,
                                 ucp_send_callback_t cb)
```

  - Operations:
    - Fetch-and-add
    - Swap
    - Compare-and-swap
    - Bitwise: AND, OR, XOR
  - Size: 32 or 64 bit

UCP API – Request Management

- Check request status:

```
ucs_status_t ucp_request_check_status(void *request)
ucs_status_t ucp_request_check_recv_status(void *request, ucp_tag_recv_info_t *info)
struct ucp_tag_recv_info {
      ucp_tag_t       sender_tag;
      size_t          length;
}
```

- Release completed request:

```
void ucp_request_free(void *request)
```

  - Must be called for every request to release it back to UCP

UCP API – Progress

- Progress communications on a worker:

  void **ucp_worker_progress**(ucp_worker_h worker)

  - Only API function which progresses communications
  - Calls non-blocking request callbacks
  - Must avoid calling it recursively

- Wait for communications without consuming CPU:

  ucs_status_t **ucp_worker_wait**(ucp_worker_h worker)

  - Returns when something may have happened on the worker
  - Need to call ucp_worker_progress after ucp_worker_wait returns.

- Get file descriptor for event loop:

  ucs_status_t **ucp_worker_get_efd**(ucp_worker_h worker, int *fd)

  - The file descriptor would be signaled in case of any event on the worker
  - Could be used in select/poll/epoll_wait

UCP API - Datatypes

- Describe the memory layout of data buffers, to avoid extra memory movements and use HW offloads when possible

- Contiguous datatype:

  ucp_dt_make_contig(_elem_size)

  - Sequence of elements of fixed size

- Scatter-gather list (IOV):

```
typedef struct ucp_dt_iov {
   void   *buffer;
   size_t  length;
} ucp_dt_iov_t;
```

  ucp_dt_make_iov()

- Generic type – user provided pack/unpack callbacks

  ucs_status_t **ucp_dt_create_generic**(const ucp_generic_dt_ops_t *ops, void *context,
  ucp_datatype_t *datatype_p)

Usage Example (1) - init

(Full example code in: test/examples/ucp_client_server.c)

```
/* Create UCP context */
ucp_context_h ucp_context;
ucp_params_t ucp_params;

ucp_params.field_mask   = UCP_PARAM_FIELD_FEATURES;
ucp_params.features     = UCP_FEATURE_STREAM;
ucp_init(&ucp_params, NULL, &ucp_context);

/* Create UCP worker */
ucp_worker_h ucp_worker;
ucp_worker_params_t worker_params;

worker_params.field_mask  = UCP_WORKER_PARAM_FIELD_THREAD_MODE;
worker_params.thread_mode = UCS_THREAD_MODE_SINGLE;
ucp_worker_create(ucp_context, &worker_params, &ucp_worker);
```

Usage example (2) - connect

```
struct sockaddr_in connect_addr;

/* Initialize destination address */
connect_addr.sin_family    = AF_INET;
connect_addr.sin_addr      = inet_addr("1.2.3.4");
connect_addr.sin_port      = htons(1234);

/* Create UCP endpoint */
ucp_ep_params_t ep_params;
ucp_ep_h ucp_ep;

ep_params.field_mask       = UCP_EP_PARAM_FIELD_FLAGS |
                             UCP_EP_PARAM_FIELD_SOCK_ADDR;
ep_params.flags            = UCP_EP_PARAMS_FLAGS_CLIENT_SERVER;
ep_params.sockaddr.addr    = &connect_addr;
ep_params.sockaddr.addrlen = sizeof(connect_addr);
ucp_ep_create(ucp_worker, &ep_params, &ucp_ep);
```

Usage Example (3) - send

```
size_t message_length;
char *message;
void *request;

/* Send stream data */
request = ucp_stream_send_nb(ucp_ep, message, message_length,
                             ucp_dt_make_contig(sizeof(char)),
                             send_completion_callback, 0 /* flags */);
if (UCS_PTR_IS_ERR(request)) {
   fprintf(stderr, "unable to send: %s\n“,
           ucs_status_string(UCS_PTR_STATUS(request));
} else if (request == NULL) {
   /* Completed */
} else {
   /* Uncompleted, need to wait for it */
   while (ucp_request_check_status(request) == UCS_INPROGRESS) {
      ucp_worker_progress(ucp_worker);
   }
   ucp_request_free(request);
}
```

Usage Example (4) - receive

```
size_t message_length, recv_length;
char *message;
void *request;

/* Receive stream data into a buffer */
request = ucp_stream_recv_nb(ucp_worker, message, message_length,
                             ucp_dt_make_contig(sizeof(char)),
                             recv_completion_callback, &recv_length, 0 /* flags */);
if (UCS_PTR_IS_ERR(request)) {
    fprintf(stderr, "unable to receive: %s\n",
            ucs_status_string(UCS_PTR_STATUS(request)));
} else if (request == NULL) {
    /* Completed */
} else {
    /* Uncompleted, need to wait for it */
    while (ucp_request_check_status(request) == UCS_INPROGRESS) {
        ucp_worker_progress(ucp_worker);
    }
    ucp_request_free(request);
}
```

Summary

- UCX is open-source, community-backed, production grade software
- Provides simple high-level communication API for RDMA and many other transports
- Optimized for performance, low overhead, low memory footprint
- Utilizes In-Network Computing technologies
- Has multiple large-scale production deployments

Unified Communication - X Framework

WEB:

www.openucx.org

https://github.com/openucx/ucx

Mailing List:

https://elist.ornl.gov/mailman/listinfo/ucx-group

ucx-group@elist.ornl.gov

ENABLER OF CO-DESIGN

Thank You

The UCF Consortium is a collaboration between industry, laboratories, and academia to create production grade communication frameworks and open standards for data centric and high-performance applications.