Designing and Developing Performance Portable Network Codes

Pavel Shamis (Pasha), ARM Research
Alina Sklarevich, Mellanox Technologies
Swen Boehm, Oak Ridge National Laboratory
Outline

• Modern Interconnect Technologies
  – Overview of existing technologies
  – Software interfaces
  – Unified Communication X Framework
• UCX programming by example
  – OpenMPI
  – OpenSHMEM
• UCX Examples
MODERN INTERCONNECT TECHNOLOGIES

Pavel Shamis (Pasha), ARM Research
Interconnects

• Interconnects are everywhere: System-on-Chip, chip-to-chip, rack, top-of-the-rack, wide area networks
Ethernet

• Used Everywhere
• Typically used in combination with TCP/UD/IP
• Socket API
• 10/25/50/100 Gb/s
• Not covered in this tutorial
• What is covered? – HPC Interconnects
Modern HPC Systems

• Not that “special” anymore
  – Commodity CPUs
  – Commodity Accelerators
  – Commodity Memories

• Still “somewhat” special
  – Form factor
  – System density
  – Cooling technologies (warm water, liquid cooling, etc.)

• The “secret sauce” – Interconnect
  – Fujitsu Tofu, IBM Torus, SGI NumaLink, Cray Aries/Gemini, TH Express-2, InfiniBand
  – Software stack – MPI + OpenMP/OpenACC

http://www.top500.org
The “secret sauce”

• Low Latency (< 1 usec)
• High Bandwidth (> 100 Gb/s)
• High injection rates (> 150M messages/sec)
• Network topologies and adaptive routing
• Scalability – efficient support for communication with millions of cores
• OS bypass (direct access to the hardware from the user level)
• Remote Direct Memory Access (avoid memory copies in communication stack)
  – Read, Write, Atomics
• Offloads
  – Collective operations, support for non-contiguous data, GPU-Direct, Peer-Direct, tag-matching, etc.
• Highly optimized network software stack (MPI + OpenMP/ACC, PGAS, etc.)
  – Low software overheads 0.6-1.2 micro-sec (MPI latency)
  – Low memory footprint (avoid O(n) memory allocations)
  – Performance portable APIs

© 2016 Unified Communication X. All rights reserved.
Advanced Semantics

- RDMA Read and Write
- Send / Receive
  - Send / Receive with TAG matching
- Atomic Operations on Remote Memory
  - SWAP
  - CSWAP
  - ADD
  - XOR
- Group Communication directives
  - Reduce, Allreduce, Scatter, Gather, AlltoAll

Socket API:
send() and recv(), or write() and read(), or sendto() and recvfrom()
## Interconnects Overview

<table>
<thead>
<tr>
<th>InfiniBand</th>
<th>RoCE</th>
<th>iWarp</th>
<th>RapidIO</th>
<th>NVIDIA NVLINK</th>
<th>Intel OmniPath</th>
<th>Bull BXI</th>
<th>Extoll</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>Open IBTA</td>
<td>Open IBTA</td>
<td>Open IETF</td>
<td>Open RapidIO</td>
<td>Proprietary</td>
<td>Proprietary</td>
<td>Proprietary</td>
</tr>
<tr>
<td>Production BW (Mb/s)</td>
<td>100Gb/s</td>
<td>100Gb/s</td>
<td>40Gb/s</td>
<td>40Gb/s</td>
<td>640Gb/s – 1600Gb/s</td>
<td>100Gb/s</td>
<td>100Gb/s</td>
</tr>
<tr>
<td>Latency (us)</td>
<td>0.6</td>
<td>0.98</td>
<td>3.4</td>
<td>&lt;1</td>
<td>&lt;1</td>
<td>&lt;1</td>
<td>&lt;1</td>
</tr>
<tr>
<td>Hardware Terminated</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Offload</td>
<td>HW/SW</td>
<td>HW/SW</td>
<td>HW/SW</td>
<td>HW</td>
<td>HW</td>
<td>HW/SW</td>
<td>HW/SW (?)</td>
</tr>
<tr>
<td>RDMA</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>?</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Market</td>
<td>HPC, Data Center</td>
<td>Data Center, HPC</td>
<td>Data Center, HPC</td>
<td>Tele, Aero, Data Center</td>
<td>HPC, Machine Learning</td>
<td>HPC, Data Center</td>
<td>HPC</td>
</tr>
</tbody>
</table>

© 2016 Unified Communication X. All rights reserved.
Typical HPC Software Stack

- **Applications**
  - CAMD, NAMD, Fluent, Lsdyna, etc.
- **Programming models**
  - MPI, UPC, OpenSHMEM/SHMEM, Co-array Fortran, X10, Chapel
- **Middleware**
  - GasNET, MXM, ARMCI, etc.
  - Part of programming model implementation
  - Sometimes “merged” with driver
- **Driver**
  - OFA Verbs, Cray uGNI, etc.
- **Hardware**
  - InfiniBand, Cray Aries, Intel OmniPath, BXI, etc.
Why we care about software stack?

- Network latency is a key
  – Sub Micro is typical for HPC Network
- Software stack overheads

© 2016 Unified Communication X. All rights reserved.
Network Programming
Interfaces (beyond sockets)

• **Open Fabric Alliance**: Verbs, Udap1, SDP, libfabrics, ...
• **Research**: Portals, CCI, UCCS
• **Vendors**: Mellanox MXM, Cray uGNI/DMAPP, Intel PSM, Atos Portals, IBM PAMI, OpenMX
• **Programming model driven**: MVAPIC1-X, GasNET, ARMCI
• **Enterprise App oriented**: OpenDataPlane, DPDK, Accelio
Vendors Specific APIs

**Pros**
- Production Quality
- Optimized for Performance
- Support and maintenance

**Cons**
- Often “vendor” locked
- Optimized for particular technology
- Co-design lags behind

© 2016 Unified Communication X. All rights reserved.
Open Source APIs

Pros
• Community (a.k.a. user) driven
• Easy to modify and extend
• Good for research

Cons
• Typically not as optimized as commercial/vendor software
• Maintenance is challenge
Research API

Pros
• Innovative and forward looking
  – A lot of good ideas for “free”

Cons
• Support, support, support
• Typically narrow focus
Unified Communication - X
Framework

UCX
History

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

Decades of community and industry experience in development of HPC software
What we don’t want to do...

Borrowed from: https://xkcd.com/927
UCX Framework Mission

- 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, Power, ARM), GPUs

Co-design of Exascale Network APIs

© 2016 Unified Communication X. All rights reserved.
A Collaboration Efforts

- Mellanox co-designs network API and contributes MXM technology
  - Infrastructure, transport, shared memory, protocols, integration with OpenMPI/SHMEM, MPICH
- ORNL co-designs network API and contributes UCCS project
  - InfiniBand optimizations, Cray devices, shared memory
- LANL co-designs network API
- ARM co-designs the network API and contributes optimizations for ARM eco-system
- NVIDIA co-designs high-quality support for GPU devices
  - GPUDirect, GDR copy, etc.
- IBM co-designs network API and contributes ideas and concepts from PAMI
- UH/UTK focus on integration with their research platforms
What’s new about UCX?

• **Simple, consistent, unified 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.
### 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.

### 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-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.

© 2016 Unified Communication X. All rights reserved.
A High-level Overview

UCX

UC-T (Hardware Transports) - Low Level API
- Transport for InfiniBand VERBs driver
- Transport for Gemini/Aries drivers
- Transport for intra-node host memory communication
- Transport for Accelerator Memory communication

UC-P (Protocols) - High Level API
- Message Passing API Domain: tag matching, rendezvous
- PGAS API Domain: RMAs, Atomics
- Task Based API Domain: Active Messages
- I/O API Domain: Stream

Applications
- MPICH, Open-MPI, etc.
- OpenSHMEM, UPC, CAF, X10, Chapel, etc.
- Parsec, OCR, Legions, etc.
- Burst buffer, ADIOS, etc.

Transport for InfiniBand VERBs driver
- OFA Verbs Driver
- Cray Driver
- OS Kernel
- Cuda

Hardware

© 2016 Unified Communication X. All rights reserved.
Clarifications

• UCX is not a device driver
• UCX is a communication framework
  – Close-to-hardware API layer
  – Providing an access to hardware’s capabilities
• UCX relies on drivers supplied by vendors
Project Management

- API definitions and changes are discussed within developers (mail-list, github, conf call)
- PRs with API change have to be approved by ALL maintainers
- PR within maintainer “domain” has to be reviewed by the maintainer or team member (Example: Mellanox reviews all IB changes)
Licensing

- BSD 3 Clause license
- Contributor License Agreement – BSD 3 based
UCX Advisory Board

- Arthur Barney Maccabe (ORNL)
- Bronis R. de Supinski (LLNL)
- Donald Becker (NVIDIA)
- George Bosilca (UTK)
- Gilad Shainer (Mellanox Technologies)
- Pavan Balaji (ANL)
- Pavel Shamis (ARM)
- Richard Graham (Mellanox Technologies)
- Sameer Kumar (IBM)
- Sameh Sharkawi (IBM)
- Stephen Poole (Open Source Software Solutions)
UCP - Protocol Layer

UC-T (Hardware Transports) - Low Level API
- Transport for InfiniBand VERBs driver
  - RC, UD, XRC, DCT
- Transport for Gemini/Aries drivers
  - GNI
- Transport for intra-node host memory communication
  - SYSV, POSIX, KNEM, CMA, XPMEM
- Transport for Accelerator Memory communication
  - GPU

UC-P (Protocols) - High Level API
- Message Passing API Domain: tag matching, rendezvous
- PGAS API Domain: RMAs, Atomics
- Task Based API Domain: Active Messages
- I/O API Domain: Stream

Applications
- MPICH, Open-MPI, etc.
- OpenSHMEM, UPC, CAF, X10, Chapel, etc.
- Parsec, OCR, Legions, etc.
- Burst buffer, ADIOS, etc.

Hardware
- OFA Verbs Driver
- Cray Driver
- OS Kernel
- Cuda

© 2016 Unified Communication X. All rights reserved.
UCP Protocol

“EVERYTHING SHOULD BE MADE AS SIMPLE AS POSSIBLE BUT NOT SIMPLER”

© 2016 Unified Communication X. All rights reserved.
Protocol Layer

• Selects the best network for the application
  – Does not have to be the same vendor
• Optimized by default
  – Protocols are optimized for the message size and underlying network semantics
  – Intelligent fragmentation
• Multi-rail, multi-interconnect communication
• Emulates unsupported semantics in software
  – No “ifdefs” in user code
  – Software atomics, tag-matching, etc.
• Abstracts connection setup
• Handles 99% of “corner” cases
  – Network out of resources
  – Reliability
  – No message size limit
  – ....and many more
UCP Objects

- **ucp_context_h**
  - A global context for the application. For example, hybrid MPI/SHMEM library may create on context for MPI, and another for SHMEM.

- **ucp_worker_h**
  - Communication and progress engine context. One possible usage is to create one worker per thread.

- **ucp_ep_h**
  - Communication peer. Used to initiate communications directives
UCP Initialization

**ucp_init** (const ucp_params_t* *params*, const ucp_config_t* *config*, ucp_context_h* *context_p*)

<table>
<thead>
<tr>
<th>in</th>
<th>config</th>
<th>UCP configuration descriptor allocated through ucp_config_read() routine.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>params</td>
<td>User defined tunings for the UCP application context.</td>
</tr>
<tr>
<td>out</td>
<td>context_p</td>
<td>Initialized UCP application context.</td>
</tr>
</tbody>
</table>

This routine creates and initializes a UCP application context. This routine checks API version compatibility, then discovers the available network interfaces, and initializes the network resources required for discovering of the network and memory related devices. This routine is responsible for initialization all information required for a particular application scope, for example, MPI application, OpenSHMEM application, etc.

Related routines: ucp_cleanup, ucp_get_version
UCP Initialization

**ucp_worker_create**

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

This routine allocates and initializes a worker object. Each worker is associated with one and only one application context. In the same time, an application context can create multiple workers in order to enable concurrent access to communication resources. For example, application can allocate a dedicated worker for each application thread, where every worker can be progressed independently of others.

Related routines: ucp_worker_destroy, ucp_worker_get_address, ucp_worker_release_address, ucp_worker_progress, ucp_worker_fence, ucp_worker_flush
UCP Initialization

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

This routine returns the address of the worker object. This address can be passed to remote instances of the UCP library in order to connect to this worker. The memory for the address handle is allocated by this function, and must be released by using `ucp_worker_release_address()` routine.
UCP Initialization

**Function**: `ucp_ep_create`

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

This routine creates and connects an endpoint on a local worker for a destination address that identifies the remote worker. This function is non-blocking, and communications may begin immediately after it returns. If the connection process is not completed, communications may be delayed. The created endpoint is associated with one and only one worker.

Related routines: `ucp_ep_flush`, `ucp_ep_fence`, `ucp_ep_destroy`
UCP Memory Management

```c
ucs_status_t ucp_mem_map (ucp_context_h context, void **address_p, size_t length, unsigned flags, ucp_mem_h *memh_p)
```

This routine maps or/and allocates a user-specified memory segment with UCP application context and the network resources associated with it. If the application specifies NULL as an address for the memory segment, the routine allocates a mapped memory segment and returns its address in the `address_p` argument. The network stack associated with an application context can typically send and receive data from the mapped memory without CPU intervention; some devices and associated network stacks require the memory to be mapped to send and receive data. The memory handle includes all information required to access the memory locally using UCP routines, while remote registration handle provides an information that is necessary for remote memory access.

Related routines: `ucp_mem_unmap`

© 2016 Unified Communication X. All rights reserved.
UCP Memory Management

### ucp_rkey_pack Function

This routine allocates memory buffer and packs into the buffer a remote access key (RKEY) object. RKEY is an opaque object that provides the information that is necessary for remote memory access. This routine packs the RKEY object in a portable format such that the object can be unpacked on any platform supported by the UCP library. In order to release the memory buffer allocated by this routine the application is responsible to call the ucp_rkey_buffer_release() routine.

**Related routines:** ucp_rkey_buffer_release

#### Function Signature

```c
ucs_status_t ucp_rkey_pack (ucp_context_h context, ucp_mem_h memh, void **rkey_buffer_p, size_t *size_p)
```

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>context</td>
<td>Application context which was used to allocate/map the memory.</td>
</tr>
<tr>
<td>memh</td>
<td>Handle to memory region.</td>
</tr>
<tr>
<td>rkey_buffer_p</td>
<td>Memory buffer allocated by the library. The buffer contains packed RKEY.</td>
</tr>
<tr>
<td>size_p</td>
<td>Size (in bytes) of the packed RKEY.</td>
</tr>
</tbody>
</table>
This routine unpacks the remote key (RKEY) object into the local memory such that it can be accesses and used by UCP routines. The RKEY object has to be packed using the ucp_rkey_pack() routine. Application code should not make any alternations to the content of the RKEY buffer.

Related routines: ucp_rkey_destroy
Communication Directives
Put

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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ep</code></td>
<td>Remote endpoint handle.</td>
</tr>
<tr>
<td><code>buffer</code></td>
<td>Pointer to the local source address.</td>
</tr>
<tr>
<td><code>length</code></td>
<td>Length of the data (in bytes) stored under the source address.</td>
</tr>
<tr>
<td><code>remote_addr</code></td>
<td>Pointer to the destination remote address to write to.</td>
</tr>
<tr>
<td><code>rkey</code></td>
<td>Remote memory key associated with the remote address.</td>
</tr>
</tbody>
</table>

© 2016 Unified Communication X. All rights reserved.
Get

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

<table>
<thead>
<tr>
<th>in</th>
<th>ep</th>
<th>Remote endpoint handle.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>buffer</td>
<td>Pointer to the local source address.</td>
</tr>
<tr>
<td>in</td>
<td>length</td>
<td>Length of the data (in bytes) stored under the source address.</td>
</tr>
<tr>
<td>in</td>
<td>remote_addr</td>
<td>Pointer to the destination remote address to write to.</td>
</tr>
<tr>
<td>in</td>
<td>rkey</td>
<td>Remote memory key associated with the remote address.</td>
</tr>
</tbody>
</table>
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 )`

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-Buffer Tag</td>
<td>R-Buffer Tag</td>
</tr>
<tr>
<td>S-Buffer Tag</td>
<td>R-Buffer Tag</td>
</tr>
<tr>
<td>S-Buffer Tag</td>
<td>R-Buffer Tag</td>
</tr>
</tbody>
</table>

© 2016 Unified Communication X. All rights reserved.
**ucs_status_ptr_t**

- **UCS_OK** - The send operation was completed immediately.
- **UCS_PTR_IS_ERR(_ptr)** - The send operation failed.
- otherwise - Operation was scheduled for send and can be completed in any point in time. The request handle is returned to the application in order to track progress of the message. The application is responsible to released the handle using `ucp_request_release()` routine.

- Request handling
  - `int ucp_request_is_completed ( void * request )`
  - `void ucp_request_release ( void * request )`
  - `void ucp_request_cancel ( ucp_worker_h worker, void * request )`
Send-Sync

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

<table>
<thead>
<tr>
<th>in</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ep</td>
<td>Destination endpoint handle.</td>
</tr>
<tr>
<td>buffer</td>
<td>Pointer to the message buffer (payload).</td>
</tr>
<tr>
<td>count</td>
<td>Number of elements to send</td>
</tr>
<tr>
<td>datatype</td>
<td>Datatype descriptor for the elements in the buffer.</td>
</tr>
<tr>
<td>tag</td>
<td>Message tag.</td>
</tr>
<tr>
<td>cb</td>
<td>Callback function that is invoked whenever the send operation is completed.</td>
</tr>
</tbody>
</table>

**Sender**
- S-Buffer Tag
- S-Buffer Tag
- S-Buffer Tag

**Receiver**
- R-Buffer Tag
- R-Buffer Tag
- R-Buffer Tag
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 )`

<table>
<thead>
<tr>
<th>in</th>
<th>worker</th>
<th>UCP worker that is used for the receive operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>buffer</td>
<td>Pointer to the buffer to receive the data to.</td>
</tr>
<tr>
<td>in</td>
<td>count</td>
<td>Number of elements to receive</td>
</tr>
<tr>
<td>in</td>
<td>datatype</td>
<td>Datatype descriptor for the elements in the buffer.</td>
</tr>
<tr>
<td>in</td>
<td>tag</td>
<td>Message tag to expect.</td>
</tr>
<tr>
<td>in</td>
<td>tag_mask</td>
<td>Bit mask that indicates the bits that are used for the matching of the incoming tag against the expected tag.</td>
</tr>
<tr>
<td>in</td>
<td>cb</td>
<td>Callback function that is invoked whenever the receive operation is completed and the data is ready in the receive buffer.</td>
</tr>
</tbody>
</table>

© 2016 Unified Communication X. All rights reserved.
Atomic Operations

- `ucs_status_t ucp_atomic_add32 ( ucpep_h ep, uint32_t add, uint64_t remote_addr, ucp_rkey_h rkey )`
- `ucs_status_t ucp_atomic_add64 ( ucpep_h ep, uint64_t add, uint64_t remote_addr, ucp_rkey_h rkey )`
- `ucs_status_t ucp_atomic_fadd32 ( ucpep_h ep, uint32_t add, uint64_t remote_addr, ucp_rkey_h rkey, uint32_t * result )`
- `ucs_status_t ucp_atomic_fadd64 ( ucpep_h ep, uint64_t add, uint64_t remote_addr, ucp_rkey_h rkey, uint64_t * result )`
- `ucs_status_t ucp_atomic_swap32 ( ucpep_h ep, uint32_t swap, uint64_t remote_addr, ucp_rkey_h rkey, uint32_t * result )`
- `ucs_status_t ucp_atomic_swap64 ( ucpep_h ep, uint64_t swap, uint64_t remote_addr, ucp_rkey_h rkey, uint64_t * result )`
- `ucs_status_t ucp_atomic_cswap32 ( ucpep_h ep, uint32_t compare, uint32_t swap, uint64_t remote_addr, ucp_rkey_h rkey, uint32_t * result )`
- `ucs_status_t ucp_atomic_cswap64 ( ucpep_h ep, uint64_t compare, uint64_t swap, uint64_t remote_addr, ucp_rkey_h rkey, uint64_t * result )`
UCT – Transport Layer

UC-T (Hardware Transports) - Low Level API
- Transport for InfiniBand VERBs driver
- Transport for Gemini/Aries drivers
- Transport for intra-node host memory communication
- Transport for Accelerator Memory communication

UC-P (Protocols) - High Level API
- Transport selection, cross-transport multi-rail, fragmentation, operations not supported by hardware
- Message Passing API Domain: tag matching, rendezvous
- PGAS API Domain: RMAs, Atomics
- Task Based API Domain: Active Messages
- I/O API Domain: Stream

Applications
- MPICH, Open-MPI, etc.
- OpenSHMEM, UPC, CAF, X10, Chapel, etc.
- Parsec, OCR, Legions, etc.
- Burst buffer, ADIOS, etc.

Hardware
- OFA Verbs Driver
- Cray Driver
- OS Kernel
- Cuda

© 2016 Unified Communication X. All rights reserved.
UCT (Transport layer) objects

- **uct_worker_h** - A context for separate progress engine and communication resources. Can be either thread-dedicated or shared.
- **uct_md_h** - Memory registration domain. Can register user buffers and/or 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.
Out-of-band Address Exchange
Memory Domain Routines

• Register/de-register memory within the domain
  – Can potentially use a cache to speed up memory registration

• Allocate/de-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

© 2016 Unified Communication X. All rights reserved.
UCT Communication Routines

• Not everything has to be supported
  – Interface reports the set of supported primitives
  – UCP uses this info to construct protocols
  – UCP implement emulation of unsupported directives

• Send active message (active message id)
• Put data to a remote memory (virtual address, remote key)
• Get data from a remote memory (virtual address, remote key)
• Perform an atomic operation on a remote memory:
  – Add
  – Fetch-and-add
  – Swap
  – Compare-and-swap

• Communication Fence and Flush (Quiet)
UCT Data Classes

• 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 values
UCT Completion Semantics

• All operations are non-blocking
• Return value indicates the status:
  – OK – operation is completed
  – INPROGRESS – operation has started, but not completed yet
  – NORESOURCE – 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 void (*uct_completion_callback_t)(uct_completion_t *self,
    ucs_status_t status);

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

struct uct_completion {
    uct_completion_callback_t func;
    int count;
};

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

typedef void *uct_mem_h;
typedef uintptr_t uct_rkey_t;

ucs_status_t uct_iface_set_am_handler(uct_iface_h iface, uint8_t id,
    uct_am_callback_t cb, void *arg, uint32_t flags);

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

ssize_t uct_ep_put_bcopy(uct_ep_h ep, uct_pack_callback_t pack_cb,
    void *arg, 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, ceil_uint64_t compare, ceil_uint64_t swap,
    uint64_t remote_addr, uct_rkey_t rkey,
    uint64_t *result, uct_completion_t *comp)
Memory

• It is a limited resource
  – The goal is to maximize the availability of memory for the application
• Avoid $O(n)$ memory allocations, where $n$ is the number of communication peers (endpoints)
• Keep the endpoint object as small as possible
• Keep the memory pools size limited
• Allocation has to be proportional to the number of in-flight-operations
Data Path

• Three main data paths:
  – Short messages – critical path
  – Medium messages
  – All the rest
Data Path / Short Messages

- Take care of the small-message case first
- Avoid function calls
- Avoid extra pointer dereference, especially store operations
- Avoid adding conditionals, if absolutely required use ucs Likely/ucs Unlikely macros
- Avoid bus-locked instructions (atomics)
- Avoid branches
- No malloc()/free() nor system calls
- Limit the scope of local variables (the time from first to last time it is used) - larger scopes causes spilling more variables to the stack
- Use benchmarks and performance analysis tools to analyze the impact on the latency and message rate
Avoid locks if possible. If needed, use spinlock, no mutex.
Reduce function calls
Move error and slow-path handling code to non-inline functions, so their local variables will not add overhead to the prologue and epilogue of the fast-path function.
Data Path / “Slow” Path

• Performance is still important
• No system calls / malloc() / free()
• It's ok to reasonable add pointer dereferences, conditionals, function calls, etc.
  – Having a readable code here is more important than saving one conditional or function call.
• Protocol-level performance considerations are more important here, such as fairness between connections, fast convergence, etc.
• Avoid O(n) complexity. As a thumb rule, all scheduling mechanisms have to be O(1).
Summary

• UCX has been integrated with:
  – MPI: Open MPI, MPICH,
  – OpenSHMEM: Reference Implementation, OSHMEM

• Support multiple transports
  – IB/RoCE: RC, UD, DCT, CM
  – Aries/Gemini: FMA, SMSG, BTE
  – Shared Memory: SysV, Posix, CMA, KNEM, XPMEM
UCX
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

© 2016 Unified Communication X. All rights reserved.
OPEN MPI INTEGRATION WITH UCX

Alina Sklarevich, Mellanox Technologies
Overview

• OMPI supports UCX starting OMPI v1.10.

• UCX is a PML component in OMPI. To enable UCX:
  – mpirun -mca pml ucx ... <APP>

• OMPI is integrated with the UCP layer.
Overview

- UCX mca parameters:
  - `pml_ucxVerbose`, `pml_ucx_priority`

- UCX environment parameters:
  - `ucx_info -f`

- For example:
  ```bash
  mpirun -mca pml ucx -x UCX_NET_DEVICES=mlx5_0:1 ... <APP>
  ```
Overview

• The calls to the UCP layer will invoke UCT calls.

• UCX will select the best transport to use.

• OMPI uses Full/Direct modex with UCX.

• UCX will connect the ranks.
UCX Features - Recap

- Tag-matching
- Remote memory operations, one sided operations
- Atomic operations
- Supported transports:
  - IB - ud, rc, dc, accelerated verbs
  - shared memory
  - uGNI
UCX Main Objects - Recap

• **ucp_context_h**
A global context for the application - a single UCP communication instance. Includes communication resources, memory and other communication information directly associated with a specific UCP instance.

• **ucp_worker_h**
Represents an instance of a local communication resource and an independent progress of communication. It contains the uct_iface_h’s of all selected transports. One possible usage is to create one worker per thread.

• **ucp_ep_h**
Represents a connection to a remote worker. It contains the uct_ep_h’s of the active transports.
UCX Main Objects - Recap

• **ucp_mem_h**
A handle to an allocated or registered memory in the local process. Contains details describing the memory, such as address, length etc.

• **ucp_rkey_h**
Remote key handle, communicated to remote peers to enable an access to the memory region. Contains an array of uct_rkey_t’s.

• **ucp_config_t**
Configuration for ucp_context_h. Loaded from the run-time to set environment parameters for UCX.
UCX API
OMPI - UCX Stack

- APPLICATION
- OMPI
- PML UCX
- UCX LIBRARY
- DRIVER
- HARDWARE
Init Stage

MPI_Init → ompi_mpi_init
mca_pml_ucx_open

ucp_config_read
ucp_init
ucp_config_release

OpenMPI
UCX

© 2016 Unified Communication X. All rights reserved.
Init Stage - Cont

mca_pml_ucx_init

ucp_worker_create

ucp_worker_get_address

ucp_worker_release_address
Init Stage - Cont

opal_progress_register(mca_pml_ucx_progress)

ucp_worker_progress
Send Flow

MPI_Isend ➔ mca_pml_ucx_isend

mca_pml_ucx_add_proc

ucp_ep_create

ucp_tag_send_nb

uct_ep_am_short
uct_ep_am_bcopy
uct_ep_am_zcopy
Send Flow - Cont

• If the send request isn’t completed
  → progress it.

• Once completed
  → callback function is invoked

• Contiguous and non-contiguous datatypes are supported.
Receive Flow

MPI_Irecv $\rightarrow$ mca_pml_ucx_irecv

ucp_tag_recv_nb

Eager Receive
Rendezvous
Receive Flow - Cont

• Expected & Unexpected queues are used

• Can probe with `ucp_tag_probe_nb`
  → `ucp_tag_msg_recv_nb`

• If the receive request isn’t completed
  → progress it.

• Once completed
  → callback function is invoked
Progress Flow

opal_progress

mca_pml_ucx_progress

ucp_worker_progress → uct_worker_progress

* Send/Receive Finished:

mca_pml_ucx_send/recv_completion
Finalization Stage

MPI_Finalize $\rightarrow$ ompi(MPI)_mpi_finalize

mca_pml_ucx_del_procs

* Per remote peer:

ucp_ep_destroy
Finalization Stage - Cont

mca_pml_ucx_cleanup

opal_progress_unregister (mca_pml_ucx_progress)

ucp_worker_destroy

mca_pml_ucx_close

ucp_cleanup
PERFORMANCE
PERFORMANCE

Setup Details:

MLNX_OFED_LINUX-3.3-1.0.0.0  
ConnectX-4  
EDR - 100Gb/sec  
Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz 
2 hosts connected via switch 

Command Line:

$mpirun -np 2 --bind-to core --map-by node -mca pml ucx  
-x UCX_TLS=rc_mlx5,cm osu_bw
osu_bw
OpenSHMEM - Overview

• PGAS Library
• One-sided Communication
• Atomic operations
• Collectives
Symmetric Memory

• All Processing Elements (PE's) share an address space (symmetric heap)
• Symmetric heap is allocated on startup
• Heapsize can be customized via environment variable SMA_SYMMETRIC_SIZE
• Symmetric data objects must be allocated with shmem_malloc
• Symmetric data objects are accessible by remote PEs
Shared global address space

• Global and static variables are symmetric objects
• Accessible by remote PEs
Initialization

shmem_init
  shmemi_comms_init
  ...
  ucp_config_read
  ucp_init
  ucp_config_release
  ucp_worker_create
  init_memory_regions (more on this later)
  ...
  ucp_ep_create (for each PE)
  ucp_config_release
  ...

© 2016 Unified Communication X. All rights reserved.
Memory registration

• OpenSHMEM registers global data (data and bss segment) and the symmetric heap
• `ucp_mem_map` maps the memory with the ucp context (returning `ucp_mem_h`)
RMA

• For RMA operations UCP needs Remote Memory Handle (remote key or rkey)
• rkeys require registered memory (ucp_mem_h)
• ucp_rkey_pack is used to generate packed representation
• The packed rkey is exchanged with remote PE(s)
• ucp_ep_rkey_unpack will return rkey_t handle
Translating symmetric addresses

- To access a remote address the rkey is needed
- Look up rkey with find_seg
- translate local buffer address into remote buffer address
RMA put

shm mem_<TYPENAME>_put
  ucx_put
    find seg
    translate_symmetric_address
  ucp_put
    uct_ep_put_short
    uct_ep_put_bcopy
    uct_ep_put_zcopy*
RMA get

shm<NAME>_get

ucx_get
  find_seg
  translate_symmetric_address

ucp_get ⇐ uct_ep_get_bcopy
  ⇐ uct_ep_get_zcopy*
RMA atomics

shmem_<TYPENAME>_<OP>

ucx_get

find_seg

translate_symmetric_address

ucp_atomic_<op,size>

uct_ep_atomic_<op,size>
PERFORMANCE

Setup:
  Turing Cluster @ ORNL
  Red Hat Enterprise Linux Server release 7.2
  (3.10.0-327.13.1.el7.x86_64)
  Mellanox ConnectX-4 VPI
  EDR IB (100Gb/s)
  Intel Xeon E5-2660 v3 @ 2.6GHz

Command Line:
  $ orterun -np 2 osu_oshmem_put_mr
OSU Message Rate

![Graph showing the relationship between message size in bytes and messages per second as per OSU Message Rate.]

© 2016 Unified Communication X. All rights reserved.
Compiling UCX

$ ./autogen.sh
$ ./contrib/configure-release --prefix=$PWD/install
$ make -j8 install
EXAMPLES
Compile OpenMPI with UCX

$ ./autgen.pl
$ ./configure --prefix=$PWD/install \  
   --with-ucx=$PWD/ucx
$ make && make install
Build OpenSHMEM on UCX

$ ./autogen.pl

$ ./configure --with-comms-layer=ucx \ 
  --with-ucx-root=${PWD}/install \ 
  --with-rte-root=${PWD}/install \ 
  --prefix=${PWD}/install

$ make && make install
UCX – Hello World Example

ucp_hello_world.c
EXAMPLES
mpirun -mca pml ucx -np 2 -H clx-orion-097,clx-orion-098 -x UCX_NET_DEVICES=mlx5_2:1 -x UCX_TLS=rc,mlx5,cm -x UCX_RNDV_THRESH=16384 --map-by node --bind-to core $OSU_TEST/osu_bw

<table>
<thead>
<tr>
<th># OSU MPI Bandwidth Test v5.0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Bandwidth (MB/s)</td>
</tr>
<tr>
<td>1</td>
<td>7.43</td>
</tr>
<tr>
<td>2</td>
<td>14.95</td>
</tr>
<tr>
<td>4</td>
<td>36.01</td>
</tr>
<tr>
<td>8</td>
<td>59.97</td>
</tr>
<tr>
<td>16</td>
<td>126.13</td>
</tr>
<tr>
<td>32</td>
<td>237.29</td>
</tr>
<tr>
<td>64</td>
<td>421.77</td>
</tr>
<tr>
<td>128</td>
<td>814.45</td>
</tr>
<tr>
<td>256</td>
<td>1292.24</td>
</tr>
<tr>
<td>512</td>
<td>2367.80</td>
</tr>
<tr>
<td>1024</td>
<td>3876.63</td>
</tr>
<tr>
<td>2048</td>
<td>6654.26</td>
</tr>
<tr>
<td>4096</td>
<td>9460.64</td>
</tr>
<tr>
<td>8192</td>
<td>16598.88</td>
</tr>
<tr>
<td>16384</td>
<td>16485.12</td>
</tr>
<tr>
<td>32768</td>
<td>11607.45</td>
</tr>
<tr>
<td>65536</td>
<td>11847.64</td>
</tr>
<tr>
<td>131072</td>
<td>11958.61</td>
</tr>
<tr>
<td>262144</td>
<td>12014.55</td>
</tr>
<tr>
<td>524288</td>
<td>12044.64</td>
</tr>
<tr>
<td>1048576</td>
<td>12058.39</td>
</tr>
<tr>
<td>2097152</td>
<td>12016.11</td>
</tr>
<tr>
<td>4194304</td>
<td>12016.12</td>
</tr>
</tbody>
</table>
mpirun -mca pm1 ucx -np 2 -H clx-orion-097,clx-orion-098 -x UCX_NET_DEVICES=mlx5_2:1 -x UCX_TLS=rc,mlx5_cmi -x UCX_RNDV_THRESHOLD=16384 --map-by node --bind-to core $OSU_TEST/osu_bibw

# OSU MPI Bi-Directional Bandwidth Test v5.0
# Size    Bandwidth (MB/s)
1         8.60
2         17.56
4         35.05
8         68.69
16        140.16
32        281.43
64        465.28
128       917.45
256       1506.33
512       2651.08
1024      4404.89
2048      7059.17
4096      11430.23
8192      14035.32
16384     17408.31
32768     26972.95
65536     21929.01
131072    22647.14
262144    23015.46
524288    23186.86
1048576   23226.71
2097152   23284.56
4194304   23191.45

© 2016 Unified Communication X. All rights reserved.
OSU_LATENCY

```
mpirun -mca pml ucx -np 2 -H clx-orion-097,clx-orion-098 -x UCX_NET_DEVICES=mlx5_2:1 -x UCX_TLS=rc_mlx5,cm -x UCX_RNDV_THRESH=16384 --map-by node --bind-to core $OSU_TEST/osu_latency

# OSU MPI Latency Test v5.0
# Size  Latency (us)
0      0.92
1      0.92
2      0.92
4      0.92
8      0.92
16     0.92
32     0.97
64     1.06
128    1.18
256    1.60
512    1.73
1024   1.95
2048   2.28
4096   2.78
8192   3.48
16384  5.74
32768  7.11
65536  9.83
131072 15.25
262144 26.12
524288 47.98
1048576 92.41
2097152 181.23
4194304 355.95
```
```
./bin/ucx_perftest vegas06 -d mlx5_0:1 -t am_bw -x rc_mlx5 -c 1

<table>
<thead>
<tr>
<th># iterations</th>
<th>typical</th>
<th>average</th>
<th>overall</th>
<th>average</th>
<th>overall</th>
<th>average</th>
<th>overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000</td>
<td>0.080</td>
<td>0.080</td>
<td>0.080</td>
<td>95.24</td>
<td>95.24</td>
<td>12483016</td>
<td>12483016</td>
</tr>
</tbody>
</table>
```
UCX
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

© 2016 Unified Communication X. All rights reserved.
Acknowledgements

This work was supported by the United States Department of Defense & used resources at Oak Ridge National Laboratory.