Collectives Offload - API

Overview

Rev. 1.3

www.mellanox.com

NOTE:

MELLANOX TECHNOLOGIES, INC. AND ITS AFFILIATES ("MELLANOX") FURNISH THIS DOCUMENT "AS IS," WITHOUT WARRANTY OF ANY KIND. MELLANOX DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT AND THOSE ARISING FROM A COURSE OF PERFORMANCE, A COURSE OF DEALING, OR TRADE USAGE. MELLANOX SHALL NOT BE LIABLE FOR ANY ERROR, OMISSION, DEFECT, DEFICIENCY OR NONCONFORMITY IN THIS DOCUMENT AND DISCLAIMS ALL LIABILITY, INCLUDING LIABILITY FOR INFRINGEMENT OF ANY INTELLECTUAL PROPERTY RIGHTS RELATED TO THE INFORMATION CONTAINED IN THIS DOCUMENT.

No license, expressed or implied, to any intellectual property rights is granted under this document. This document, as well as the software described in it, are furnished under a separate license and shall only be used or copied in accordance with the terms of the applicable license. The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as any commitment by Mellanox. Except as permitted by the applicable license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Mellanox.

Names and logos identifying products of Mellanox in this document are registered trademarks or trademarks of Mellanox. Voltaire is a registered trademark of Mellanox Technologies, Ltd. All other trademarks mentioned in this document are the property of their respective owners.

Mellanox Technologies, Inc.
350 Oakmead Parkway Suite 100
Sunnyvale, CA 94085
U.S.A.
www.mellanox.com
Tel: (408) 970-3400
Fax: (408) 970-3403

Mellanox Technologies Ltd
PO Box 586 Hermon Building
Yokneam 20692
Israel
Tel: +972-4-909-7200
Fax: +972-4-959-3245

Mellanox®, BridgeX®, ConnectX®, CORE-*Direct*®, InfiniBlast®, InfiniBridge®, InfiniHost®, InfiniRISC®, InfiniScale®, InfiniPCI®, PhyX®, Virtual Protocol Interconnect and Voltaire are registered trademarks of Mellanox Technologies, Ltd.

FabricIT, MLNX-OS and SwitchX are trademarks of Mellanox Technologies, Ltd.

All other trademarks are property of their respective owners.

Contents

1 Introduction

To meet the needs of scientific research, supercomputers are growing at an unrelenting rate. As supercomputers increase in size from mere thousands to hundreds-of-thousands of processor cores, new performance and scalability challenges have emerged. In the past, performance tuning of a parallel application could be accomplished fairly reliably by separately optimizing its algorithms, communication, and computational aspects. However, as we continue to scale future larger machines, these issues become co-mingled and must be addressed comprehensively.

Collectives communication, which have a crucial impact on the application's scalability, are frequently used by scientific simulation codes like broadcasts for sending around initial input data, reductions for consolidating data from multiple sources and barriers for global synchronization. Any Collectives communication executes some global communication operation by coupling all processes in a given group. This behavior tends to have the most significant negative impact on the application's scalability. In addition, the explicit and implicit communication coupling, used in high-performance implementations of Collectives algorithms, tends to magnify the effects of system-noise on application performance, further hampering application scalability.

Mellanox ConnectX®-2 addresses the Collectives communication scalability problem by offloading a sequence of data-dependent communications to the Host Channel Adapter (HCA). This solution provides the hooks needed to support computation and communication overlap, allowing the communication to progress asynchronously in hardware while at the same time computations are processed by the CPU. It also provides a means to reduce the effects of system noise and application skew on application scalability.

The Message Passing Interface (MPI), a ubiquitous communication library, which is in use by scientific applications, defines fifteen blocking collectives operations. Currently, non-blocking collectives operations implemented in ConnectX-2 are going under standardization processes for the MPI-3 software version.

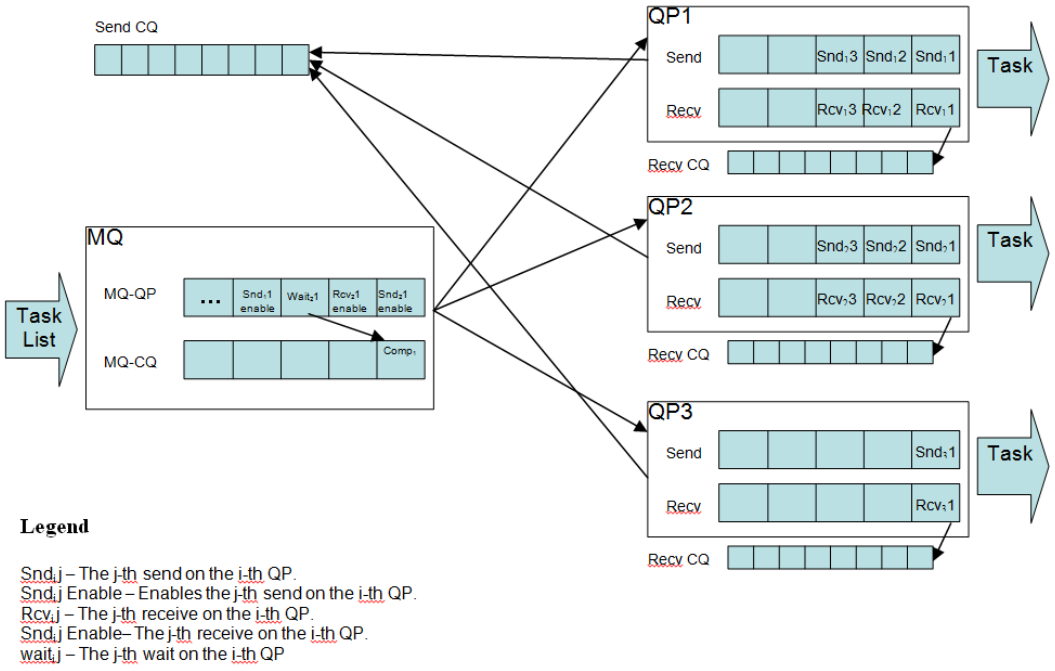
The ability to offload communication patterns to hardware is naturally well-suited to allow overlap of non-blocking collectives operations with application computation. Offloading communication processing to the HCA provides the capability to minimize the effects of system noise and to increase the communication–computation overlapping significantly.

4 Management Queues (MQs)

4.1 Overview

In order to relieve the CPU from the communication management workload, the ConnectX-2 Core-Direct includes support for wait, calc (a new feature of send WQEs), send-enable and receive-enable work requests that MPI collectives use for network communication. ConnectX2 introduces new types of queue pairs for managing this communication: Manage Queue Pair (MQ-QP) and Manage Completion Queue Pair (MQ-CQ). The Manage Queues are software feature for using the libmqe.

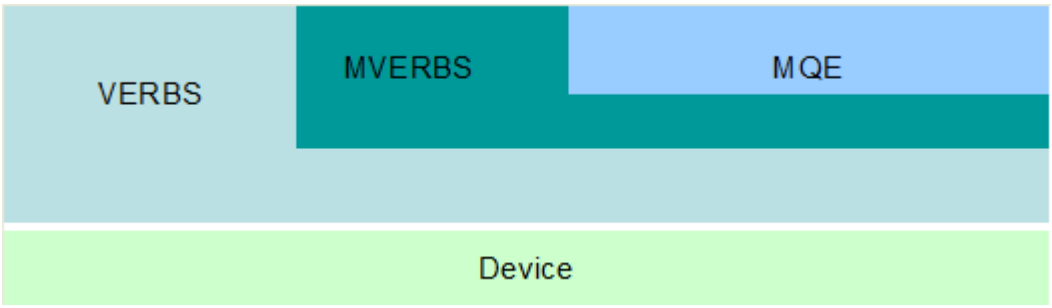
MQ-QP holds management work requests that instruct when to execute send or receive work requests. MQ-CQ holds the completion status for a series of network operations (send, receive, wait).



5 API

A Mellanox API, supporting the new ConnectX-2 hardware capabilities, is implemented by two libraries: LIBMQE (MQE API) and LIBMVERBS (MVERBS API). MQE is an interface for the new management queue including MQ creation, destruction, post of a task list and wait capabilities.

MVERBS implements the ConnectX-2 extended interface. It exposes new basic primitives of ConnectX-2 device such as Calc operation type, supported data types by Calc, MQ work request operations etc.



The MQE and MVERBS API are under development and subject to changes.

5.1 MQE API

5.1.1 Create MQ Context

```
struct mqe_context *mqe_context_create(struct i bv_context *ibv_ctx,
                                       struct ibv_pd *pd,
                                       struct mqe_context_attr *attr);
```

This struct creates a Management Queue (MQ) context object representing an MQ-QP and the corresponding MQ-CQ. The MQ-QP is associated with the protection domain pd. The argument attr is a mqe_context_attr struct, as defined in <infiniband/mqe.h>.

The created MQ-CQ is attached to the MQ-QP and is used both as send and receive CQ.

If attr.cq is set, the MQ-CQ will be the CQ it points to. If attr.cq is NULL, a new CQ will be created and used as the MQ-CQ.

5.1.2 Destroy MQ Context

```
int mqe_context_destroy(struct mqe_context *ctx);
```

Destroys the mqe_context ctx.

If the MQ-CQ was created by the driver, upon calling mqe_context_create, the MQ-CQ is destroyed. Otherwise it's up to the user, who created the CQ, to destroy it.

5.1.3 Post Task

```
int mqe_post_task(struct mqe_context *ctx,
                                    struct mqe_task *task_list,
                                     struct mqe_task **bad_task);
```

This struct posts the linked list of MQ tasks, starting with task_list, to the MQE ctx.

It stops processing tasks from this list at the first failure, and returns this failing task through bad_task, if it is not NULL.

Tasks can be posted to one or more communication QPs and / or to the MQ-Q.

Send / receive tasks are posted to the list of communication QPs stated in task_list->post.qp.

Wait tasks are posted to communication QPs and / or to the MQ-QP, according to the following semantics:

If task_list->wait.mqe_qp is NULL the wait task is posted to the MQ-QP.

If task_list->wait.mqe_qp is not NULL, wait tasks are posted to the list of communication QPs stated in task_list->wait.mqe_qp. If MQE_WR_FLAG_WAIT_ON_MQE is set in task_list->flags, the wait task will also be posted to the MQ-QP.

5.1.4 Post Task

```
int mqe_post_task(struct mqe_context *ctx,
                  struct mqe_task *task_list,
                  struct mqe_task **bad_task);
```

This struct posts a list of work requests; each of them specifies an operation opcode, a request flag and a post/wait/enable task for execution.

It stops processing tasks from this list at the first failure, and returns this failing task through bad_task, if it is not NULL.

Tasks can be posted to one or more communication QPs and / or to the MQ-Q.

`send/receive` tasks are posted to the list of communication QPs stated in `task_list->post.qp`.

`wait` tasks are posted to communication QPs and / or to the MQ-QP, according to the following semantics:

- If `task_list->wait.mqe_qp` is NULL the wait task will be posted to the MQ-QP
- If `task_list->wait.mqe_qp` is not NULL, wait tasks will be posted to the list of communication QPs stated in `task_list->wait.mqe_qp`
- If `MQE_WR_FLAG_WAIT_ON_MQE` is set in `task_list->flags`, the wait task will also be posted to the MQ-QP

5.1.5 Poll MQ Context

```
int  mqe_poll_context(struct mqe_context *ctx, struct mqe_wc *wc);
```

Polls the MQ-CQ of MQE ctx for a single work completion.

The argument wc is a pointer to an object of type ibv_wc structs, as defined in <infiniband/mqe.h>, and is filled if a completion was polled.

In case of success the function returns 1 (if a completion was polled from the MQ-CQ) or 0 (if no completions were polled).

In case of failure the function returns a negative number.

5.2 MVERBS API

5.2.1 Query Device

```
int ibv_m_query_device(struct ibv_context *context,
                       struct ibv_m_dev_attr *device_attr);
```

This struct returns the attributes of the device with context.

The argument device_attr is a pointer to an ibv_m_dev_attr

The struct, as defined in <infiniband/mverbs.h>, is constructed from ibv_device_attr, as described in <infiniband/verbs.h> and special device capability flags, added for CORE-Direct support.

5.2.2 Query Calc Capabilities

```
int ibv_m_query_calc_cap(struct ibv_context *context,
 enum ibv_m_wr_calc_op calc_op,
 enum ibv_m_wr_data_type data_type,
 int *operands_per_gather,
 int *max_num_operands);
```

This struct returns true if an input calc operation on an input data type is supported by the device. O/W returns false. Note that some operations are not supported by the device but are supported by the driver, therefore pack_data_for_calc and unpack_data_from_calc functions should be used, as described below.

The argument calc_op is of type enum ibv_m_wr_calc_op and the argument data_type is of type enum ibv_m_wr_data_type, as defined in <infiniband/mverbs.h>.

If the operation is supported the number of supported operands per gather element and the maximum number of operands per calc operation are returned in arguments operands_per_gather and max_num_operands, respectively.

5.2.3 Post Send

```
int ibv_m_post_send(struct ibv_qp *qp,
                                  struct ibv_m_send_wr *wr,
                                  struct ibv_m_send_wr **bad_wr);
```

This struct posts the linked list of work requests (WRs) starting with wr to the send queue of the queue pair qp. The WRs are of type struct ibv_m_send_wr , as defined in <infiniband/mverbs.h>.

The function stops processing WRs from this list at the first failure (that can be detected immediately while requests are being posted), and returns this failing WR through bad_wr.

If the send inline flag is set, by setting IBV_SEND_INLINE in wr->send_flags, the data buffers used for the WR may be reused immediately after the call returns.

This function corresponds to function ibv_post_send, as described in <infiniband/verbs.h>, but allows posting WRs with special attributes for CORE-Direct.

5.2.4 Pack Data for Calc

```
int pack_data_for_calc(struct ibv_context *context,
enum ibv_m_wr_calc_op op,
enum ibv_m_wr_data_type type,
bool host_buffer_in_net_order,
const void *host_buffer,
uint64_t id,
enum ibv_m_wr_calc_op *out_op,
enum ibv_m_wr_data_type *out_type,
void *network_buffer);
```

This struct modifies a host input buffer to suite the format required by Calc operation.

It performs required data manipulations on the input buffer such as bytes order conversion (from host to network), resulting with formatted data for calc.

The output network buffer must be 16 bytes aligned and is allocated as a 64 bit array except for the cases in which the operation is MINLOC or MAXLOC, where the buffer is allocated as a 128 bits array.

The packing procedure may change the operation or the data type to be given to Calc. Therefore the parameters out_op and out type will correspondingly hold the operation to be performed and the data type to perform the operation on.

5.2.5 Unpack Data from Calc

```
int unpack_data_from_calc(struct ibv_context *context,
enum ibv_m_wr_calc_op op,
enum ibv_m_wr_data_type type,
const void *network_buffer,
uint64_t *id,
void *host_buffer);
```

This struct modifies Calc's output buffer, represented in network byte order, to suite the format expected by a host. Required data manipulations such as byte order conversion (from network to host) are performed on the data buffer, resulting with the formatted data expected by the host.

The output host buffer format is to be set according to the expected data type as indicated by 'type', and is assumed to be big enough to contain the entire data.

5.2.6 Get Actual Calc Op

```
enum ibv_m_wr_calc_op get_actual_calc_op(enum ibv_m_wr_calc_op op);
```

It returns the actual Calc operation to be performed by the device after executing the packing procedure on the host data.

5.2.7 Get Actual Calc Data Type

```
enum ibv_m_wr_data_type  get_actual_calc_dtype(enum ibv_m_wr_data_type type);
```

It returns the actual data type to perform Calc operation on after executing the packing procedure on the host data.

The *actual operation* and the *actual data type* are both parameters returned by pack.