A uGNI-Based MPICH2 Nemesis Network Module for Cray XE Computer Systems

Howard Pritchard Igor Gorodetsky

Cray Inc.

Abstract

Recent versions of MPICH2 have featured Nemesis - a scalable, high-performance, multi-network communication subsystem. Nemesis provides a framework for developing Network Modules (Netmods) for interfacing the Nemesis subsystem to various high speed network protocols. Cray has developed a User-Level Generic Network Interface (uGNI) for interfacing MPI implementations to the internal high speed network of Cray XE and follow-on compute systems. This paper describes the design of a uGNI Netmod for the MPICH2 nemesis subsystem. MPICH2 performance data on the Cray XE will be presented. Planned future enhancements to the uGNI MPICH2 Netmod will also be discussed.

1 Introduction

The Cray XE represents a fundamental change in network architecture from its predecessor XT systems [1]. The Cray XE *Gemini* network provides user–space applications with a low-overhead, programmed I/O (PIO) mechanism for accessing memory on remote nodes in a true one–sided fashion. Termed *Fast Memory Access* (FMA), this hardware supports remote direct memory access (RDMA) read, write, and atomic memory operations (AMOs) to memory at remote nodes. The Gemini also has a Block Transfer Engine (BTE) to offload RDMA read and write operations from the host processor. In addition, Gemini was designed with fault–tolerance related features that allow software to recover from various network errors more reliably than on the predecessor XT system.

Reflecting these fundamental differences between the two networks, the Message Passing Interface (MPI) MPICH2 implementation deployed on XE systems is substantially different from the Portals–based MPICH2 deployed on predecessor XT systems.

The intent of this paper is to provide an overview of the MPICH2 uGNI Network Module that interfaces to the Cray XE, the software layer it uses to interface to the underlying network, as well as other support software the Network Module utilizes to obtain maximum possible performance. The paper is organized as follows: important elements of the Generic Network Interface (GNI) are presented, an overview of MPICH2 Nemesis and its Network Module framework follows, after which details of the uGNI Network Module and related support software are presented. Some basic performance data on Cray XE is presented. The paper ends with a discussion of future work planned for the uGNI Network Module.

2 Generic Network Interface API Overview

The Generic Network Interface (GNI) provides a low–level API for network middleware to efficiently utilize the Cray XE network. GNI is primarily intended for user–space and kernel-space network applications whose communication patterns are message–based in nature, and where the ability to recover from network faults is of importance. The API is not intended for use by End-User applications. GNI is also not optimal for middleware which supports applications like Partitioned Global Address Space (PGAS) compilers and other applications requiring high performance for very fine–grain remote memory access.

A layered approach was taken in designing GNI. A lowest level Generic Hardware Abstraction Layer (GHAL) is used to interface to particular instantiations of Gemini. This layer is used to mask specific details of hardware registers etc. from the upper level components. Other components of the GNI stack include *kGNI* - the upper level device driver which also implements the kernel level GNI API, and a *uGNI* library which implements the API for user-space applications. Although the software is layered, it should be pointed out that core components are shared between the kernel (kGNI) and user–space (uGNI) interfaces. This approach facilitates development, maintenance, and testing. For example, changes can be made to a core component and first tested in user–space MPI testsuites and applications before being incorporated into kernel level services like the Lustre's LNET transport layer. Likewise, fault tolerance features, which may be easier to stress test using kernel–level components can then be reused in user–space without extensive additional testing. Figure 1 depicts the layered view of the GNI software stack with two sample clients – the MPICH2 in user–space, and GNILND (Lustre LNET) in kernel space. The various major

components of the API, how they map to the XE hardware, and typical software usage models are described below.

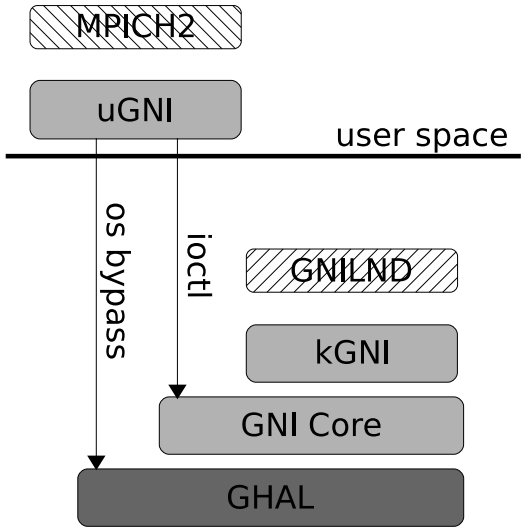

Figure 1. GNI Software Stack with MPICH2 and LNET(GNILND) example clients

2.1 Communication Domains

Since the Gemini allows for remote direct memory access from user-space, a hardware protection mechanism exists to validate all remote memory access requests generated by applications. To utilize the hardware protection mechanism, GNI provides software with a *Communication Domain* (CDM) construct. Processes (*Peers*) use a previously agreed upon *Protection Tag* (ptag) to define and join a CDM. For example, the GNI–based Lustre file system uses a ptag reserved for it system wide. Each peer must also supply a 32-bit *inst id* which is unique within the CDM. For user space applications, the ALPS [4] job launch system provides a ptag value for each job. On production XE systems, user–space applications are not permitted to choose arbitrary ptag values.

GNI provides additional privileged functions specifically for ALPS to use when launching a job, including functions for configuring the Gemini Node Translation Table (NTT), and various NIC resource limits. The NTT provides greater flexibility in allocation of ptags for user space applications. The usage of the NTT is transparent to user-space software using the GNI API.

A *modes* argument supplied to the interface which creates the CDM can be used to specify additional characteristics of the CDM. Examples are options for handling Copy–on–Write (COW) problems that can occur when a process which has registered memory with the Gemini does a *fork*, how the kGNI driver should handle NIC errors that can be traced back to the application, etc.

2.2 NIC Handles

The CDM is essentially a software construct, and cannot be used directly for launching network transactions. Thus, prior to initiating any communication with other peers, a process must first attach a CDM to a Gemini Network “Card” (NIC). This is somewhat analogous to the IP socket *bind* operation. A GNI *NIC Handle* is returned to the application as part of this attach operation. A CDM can be attached to multiple Gemini NICs, i.e. multiple NIC handles can be associated with a given CDM. By taking this approach, GNI can be used both for current XE systems where there is only a single Gemini NIC per node, as well as allowing for the flexibility to support multiple NICs per node in possible future system configurations.

2.3 End Points

Once a process has obtained a NIC handle by binding a CDM to a Gemini, *End Points* can be created. GNI uses the End Point (EP) construct for managing data exchange between peers within a CDM. This construct is not directly related to any Gemini resources, but is useful for resource management associated with network transactions. An EP can be bound to a given Gemini *Nic Address* and *Remote Inst Id*—the unique 32-bit identifier provided by the peer when it joined the CDM. EPs can be used for session management (see Section 2.4), sending messages to a remote peer, and/or RDMA transactions. The API provides a function for unbinding an EP from a NIC address and remote inst id. For EPs that are used exclusively for session management setup and/or RDMA transactions, unbinding an EP and rebinding the EP to a new NIC address/remote inst id is a lightweight operation. In these cases, the unbind operation does not involve communication with a remote peer or system calls.

2.4 Session Management

GNI provides session management functions to support network applications which manage messaging channels (Section 2.8) between peers dynamically. Peers can exchange information concerning channel setup using these session management functions. GNI also supports the notion of *wildcard* datagrams which can be used, for example, by client/server models in which the server peers wait for incoming connection requests from client peers.

2.5 Memory Registration

In order for the Gemini to directly access a memory region on a remote node, the memory region at the remote node must

have been previously registered with the Gemini at the remote node. Also, in order for the Gemini's BTE and FMA hardware to access a local memory region, the memory region must be registered with the local Gemini. GNI provides memory registration interfaces for applications to register memory with the Gemini. The interface allows for the application to specify access permissions and memory ordering requirements with respect to the local processor interface. It also enables the application to associate a *Completion Queue*(CQ) with the registered memory region. GNI returns an opaque *Memory Handle* (MH) structure to the application upon successful invocation of one of the memory registration functions. The MH can then be used for RDMA transactions and messaging.

2.6 Completion Queue Management

Gemini *Completion Queues* (CQ) provide a light-weight notification mechanism for software to determine when

- the data in a RDMA Write or non-fetching AMO transaction has been delivered to the target Gemini,
- the result data for RDMA Read or fetching AMO transaction has been delivered into local memory,
- a message from a remote peer has been delivered to a previously registered local memory region.

GNI provides an interface for creating and destroying CQs. Generally software uses a *"TX" CQ* for the first two types of notifications, and a separate *"RX" CQ* for the third type. For RDMA transactions using the FMA hardware, the application binds a CQ to a NIC handle. When using the BTE for RDMA transactions an application can specify the CQ in the *Post Descriptor* defining the transaction (see Section 2.7). An application can bind a CQ to a registered memory region to receive notifications for messages arriving in that memory region.

An application can check for presence of *Completion Queue Events* (CQEs) on a CQ in either polling or blocking mode. A CQE includes application specific data, information about what type of transaction is associated with the CQE, and whether or not the transaction associated with the CQE completed successfully or not. Sufficient information is provided in the CQE error field to allow an application to discriminate between errors arising due to transient errors in the network or target node of the transaction, and those related to software errors at the source or target.

2.7 Remote Direct Memory Access Transactions

GNI provides an interface for initiating RDMA transactions using either the Gemini BTE or FMA hardware. To initiate an RDMA transaction, the application takes the following steps. First, the application binds an EP to the remote NIC address and remote inst id that is the target of the transaction. A *post descriptor* is created with information about the transfer including pointers to source and target buffers, the required MHs, etc. This descriptor is then posted to the EP. The particular function used to post the descriptor to the EP determines whether or not the BTE or FMA hardware will be used to handle the transaction.

The BTE is generally preferred for moving all bulk message data as it offloads the work of moving the data from the host processor to the NIC. Also, on XE systems, the BTE is generally more efficient at moving data through the node's internal coherent HyperTransport interconnect as well. The FMA hardware is best used for shorter control data and AMOs. The FMA hardware may also give better results when there are many processes on the node trying to issue medium size (2–8 KB) RDMA transactions in a synchronized, bursty fashion. See Section 5.2.

2.8 Messaging Facility

The Gemini provides a specialized RDMA Write with remote notification operation which is used by GNI to provide a messaging API for applications. For user-space applications, there are two types of messaging facilities available.

The GNI *Short Message* (SMSG) facility provides the highest performance in terms of latency and short messages rates, but comes at the expense of memory usage, which grows linearly with the number of peer-to-peer connections. Two peers in a CDM can establish a SMSG channel using the following procedure:

- Each peer creates an EP and binds it to the NIC address and remote inst id of the peer with which an SMSG channel is to be established
- Each peer then creates a *mailbox* for its end of the SMSG channel by allocating memory and registering it with the Gemini using functions described in Section 2.5. The resulting MH and other info is used to initialize a SMSG channel *attributes* structure.
- The peers exchange these attributes using the Session Management methods described above in Section 2.4.
- The pairs then initialize the SMSG channel using their local attributes structure and the remote attributes structure obtained from the peer. Messages can now be exchanged between peers of the SMSG interface.

GNI also provides a *Message Queue* (MSGQ) facility that user-space applications may use. MSGQs provide lower performance, particularly in terms of short message rate, but are much more scalable in terms of memory usage than SMSG channels. Memory usage scales as the number of *nodes* in the job rather than peers. Setup of MSGQs is similar to that

described above for SMSG connections, but is done on a per-node rather than per-peer basis, with many of the steps being hidden within the uGNI layer. The maximum size message that can be sent using the MSGQ facility is 128 bytes. In theory SMSG can be used to deliver messages up to 16MB in size, but owing to memory footprint constraints and performance considerations, the practical upper limit is in the kilobyte range. For both facilities, messages are delivered in order.

Both the SMSG and MSGQ facilities exploit design features of the Gemini write with remote notification hardware to implement a reliable messaging protocol. Using this mechanism, the target Gemini of an incoming message will only deliver the notification flag if the message was successfully received. At the sender, or initiator Gemini, a *TX CQE* is generated for each message sent. The error field component of the CQE will indicate whether or not

- the message was successfully received at the target Gemini
- a network timeout or other transient error occurred either in the delivery of the message to the target node, or the delivery of the hardware responses back from the target Gemini to the initiator Gemini
- a software error related problem occurred at the receiver, such as a process *segfault* that was to receive the message

The GNI SMSG and MSGQ software makes use of the error, if any, reported in the *TX CQE*, in conjunction with a sliding window protocol, to determine whether a message needs to be retransmitted or has been successfully received at the target node.

The GNI API provides a function for applications to use to determine whether or not the error reported in the *TX CQE* is a transient, and thus recoverable, error.

2.9 Asynchronous Error Reporting

GNI provides an asynchronous error notification interface. This interface can be used in conjunction with the error notifications provided in CQEs to enhance error reporting and to facilitate debugging of application and system problems.

3 MPICH2 NEMESIS

MPICH2 is a widely used, open–source implementation of MPI developed and maintained by Argonne National Laboratory (ANL) [6]. The software is implemented in a layered fashion depicted in Figure 2. At the top level of MPICH2 is an Abstract Device Interface - version 3 (ADI3) device. It presents an MPI–2.2 compliant interface to applications, while presenting the ADI3 interface to the *device* layer below it.

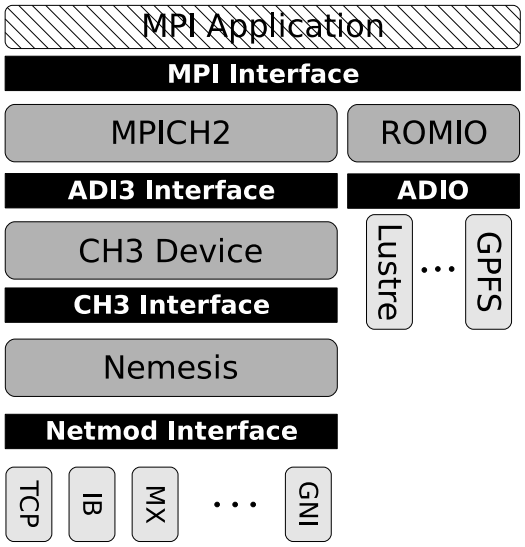

Figure 2. MPICH2 and Nemesis CH3 channel software stack with sample Network Modules

Network hardware vendors can choose to port MPICH2 to a custom interconnect by implementing an ADI3 device. This approach may allow for better utilization of a custom network. This was the approach taken for the port of MPICH2 to the Cray XT. However, developing and maintaining a complete, custom ADI3 device can be very expensive, and frequently leads to redundant development that contributes little in the way of differentiation of a custom interconnect. Additionally, the pace of development of MPICH2 at Argonne has accelerated recently, partially driven by the desire to implement proposed MPI–3 extensions to MPI.

An alternative to developing a full ADI3 device is to implement a *channel* for the CH3 ADI3 device that comes with MPICH2. In principle this interface requires significantly less development effort to code to, while still delivering reasonable performance. However, as the availability of commodity RDMA capable networks increased, and thus the interest in porting MPICH2 to networks using protocols other than TCP sockets, deficiencies with the channel API became apparent. Vendors and other organizations more often than not just reworked the CH3 ADI3 device for a particular network, rather than using the channel interface.

This was one of the motivations for the introduction of a new *Nemesis* channel to the CH3 ADI3 device. The goals of the Nemesis channel, as stated by the authors, are scalability, high performance intra–node communication, high performance internode communication, and multi–network inter–node communication [2]. Although the ultimate goal is to make Nemesis a self–standing ADI3 device, it was found sufficient at the time to make modifications to the CH3 device itself to better support the Nemesis package as a channel.

The major components of Nemesis are a highly optimized on-node messaging system and a multi–method capable framework for implementing *Network Modules* (Netmod) within Nemesis. The framework is flexible and can be used for a variety of interconnects as evidenced by existing modules such as Myrinet MX and GM and a recent IB module available in the MVAPICH2 version of MPICH2 [8]. The basic function of a Netmod is to move control messages — which can be application messages — and data across a network. The upper components of Nemesis implement the MPI portion, e.g. message matching, handling of unexpected messages, etc. There are hooks in Nemesis to support Netmods that have MPI–awareness such as hardware message matching in their networks.

Nemesis uses a callback–function approach to interface with Netmods. Callback functions can be roughly divided into three sets. One set of callbacks is associated with the Netmod as a whole. Some of the most important of these are

- **Initialization** - does any initialization specific to the Netmod. It is invoked by Nemesis as part of the MPI_Init procedure.
- **Finalization** - does any finalization specific to the Netmod. It is invoked by Nemesis as part of the MPI_Finalize procedure.
- **Checkpointing** - does any checkpoint specific activities required by the Netmod. It is invoked when a job is being checkpointed. Note MPICH2 uses the BLCR package for checkpointing.
- **Restart** - does any restart specific activities required by the Netmod. It is invoked when a previously checkpointed job is being restarted.
- **Virtual Connection Initialization** - does any Netmod specific step required to initialize the *virtual connection* (VC) structure used by the CH3 device to manage messaging between ranks in the job. This step does not necessarily mean the VC is *active*. Additional steps may be needed to send messages over the VC.
- **VC terminate** - does any connection termination specific to the Netmod for a given VC. This is currently invoked by Nemesis at MPI_Finalize, or when disconnecting from a group started using MPI-2 process creation functions.
- **VC destroy** - does any Netmod specific steps required to clean up resources associated with a VC. This is currently invoked by Nemesis at MPI_Finalize, or when disconnecting from a group started using MPI-2 process creation functions.

The second set of callbacks are associated with the VC structure. The most important are

- **maximum eager message size** - this is not a callback function, but a Netmod sets this value to control when Nemesis switches from an eager to rendezvous protocol for the VC. Note that this field is ignored for certain kinds of transfers.
- **iStartContigMsg** - the Netmod callback Nemesis invokes to enqueue a message for sending on a VC
- **iSendContig** - an optimized version of iStartContigMsg that potentially allows the Netmod to avoid initializing an MPI request structure
- **pause_send_vc** - callback function for Nemesis to invoke when pausing or quiescing a VC at checkpoint time
- **restart_vc** - callback function for Nemesis to invoke when restarting a VC during the restart procedure for a checkpointed job
- various callbacks associated with the **Long Message Transfer Protocol**

A third set of callbacks is associated with a *Communication Operations* structure defined by the Netmod. This can be used for Netmods which elect to implement higher level MPI functions like MPI_Isend, etc. directly. These callbacks are not currently used by the uGNI Netmod and so are not discussed further here.

A Netmod invokes the *MPID_nem_handle_pkt* function to deliver data coming off the network up into Nemesis on the receive side. This function takes as arguments the VC associated with the message data, a pointer to the data coming off the wire, and the length of the data. Nemesis expects this data to be delivered in the same order it was sent from the sending rank. Nemesis can handle receiving messages as fragments. Upon return from *MPID_nem_handle_pkt* the Netmod can assume its safe to reuse any buffers or resources associated with the data packet.

Nemesis features a *Long Message Transfer* (LMT) protocol that facilitates implementation of zero-copy transfers for Netmods interfacing to networks that support these types of operations. If a Netmod defines the LMT callbacks on its VC's, then Nemesis will use that method for sending messages larger than the rendezvous threshold. When the LMT path is used, only short control messages actually move through the Nemesis stack itself. The bulk message data can be transferred in a zero–copy fashion from the application's send buffer into the application's receiver buffer. Note there are some exceptions to when the LMT is used, even when the message size is greater than the eager message size defined in the VC struct. Ready send messages do not use the LMT path. Messages generated from MPI-2 RMA functions do not currently use this path.

4 The uGNI Netmod

The Gemini NIC has a number of characteristics which enable good MPI performance. The most significant is the FMA hardware, which provides a low–overhead, os–bypass pathway for injecting short messages into the network. This enables the MPICH2 on Cray XE to realize much lower latencies and vastly higher message rates than with the predecessor XT systems. The BTE allows for offloading of bulk data movement from the host processor, providing one of the components essential for realizing independent progress of MPI messages. Other factors which significantly influenced the design of the uGNI Netmod were the requirement to be able to handle transient network faults, interoperability with other program models, reuse of as much of the existing infrastructure in MPICH2 as possible, and extensibility to support at least some of the proposed MPI-3 Fault Tolerance features [3].

4.1 Initialization

The uGNI Netmod's initialization method is invoked by Nemesis as part of the overall MPI initialization procedure that takes place when an application calls *MPI_Init* or *MPI_Init_thread*. A CDM is created using the *ptag* value supplied by ALPS. In addition to the *ptag*, ALPS also supplies the list of Gemini interfaces that the ranks of the job on the local node can use. Although it is not anticipated that the Cray XE will support multiple Geminis NICs per node, the uGNI netmod and ALPS support infrastructure was coded to be able to use multiple Gemini NICs. The Netmod then attaches the CDM to all available Gemini NICs. The resulting NIC handles are then used for the following steps.

The Netmod next initializes a registration cache (see Section 4.7), TX and RX CQs are created using the NIC handles, DMA buffers are registered with the NIC handles, and a freelist of transaction management structures is created. The transaction management structures serve multiple purposes: to avoid overflow of the TX CQ by limiting the total number of transactions the Netmod has outstanding at any point in time, associating CQEs pulled off the TX CQ back to the original transaction, and for holding state and data required to handle transient network failures.

Depending on which GNI interface is selected at run time for sending short messages – SMSG or MSGQ – two different steps are taken during initialization of the Netmod.

If the SMSG method is selected, an initial set of SMSG *mailboxes* is created and registered with the NIC handles. A set of EPs are created in order to post *wildcard* datagrams with each of the NIC handles. If dynamic connections are enabled – the default – no further steps are taken during initialization of the Netmod. If dynamic connections are disabled, the Netmod registers a callback with Nemesis to run the Connection Setup phase to connect all ranks prior to return from *MPI_Init* to the application.

When using the GNI MSGQ facility, it is simpler to build all of the per–node connections prior to returning from *MPI_Init*. In this case, the Netmod initialization function registers a callback with Nemesis to run connection setup prior to returning from *MPI_Init* to the application.

The Netmod returns a MPICH2 CH3 *business card* (BC) to Nemesis as part of the Netmod initialization procedure. Upon return from the Netmod's initialization callback function, Nemesis commits the BC to the Process Manager Interface's (PMI) key–value space (KVS).

As the last stage of the overall Nemesis channel initialization, Nemesis runs any callbacks that the Netmod's initialization routine had registered.

4.2 Connection Setup

When running in dynamic connection mode, SMSG channels are only established when a given rank in the job needs to send a message to another rank. If a channel has not been established yet, the sender allocates an SMSG mailbox, and prepares a channel establishment message describing the mailbox location within the pool of registered memory from which the mailbox was allocated. This channel establishment message is then sent via the GNI session management (Section 2.4) protocol to the target remote NIC address and remote inst id. The remote NIC address is obtained from the PMI KVS. In most cases, one of the intended receiver's *wildcard* datagrams is matched with this channel establishment message. The *wildcard* datagram contains channel information about any mailbox the receiver has prepared for handling incoming connection requests. As part of the session management code within the kGNI driver, the *wildcard* datagram, after having been matched, is sent back to the sender. As the sender and receiver return into MPI, they dequeue the completed datagram sessions and use the information to complete the SMSG channel. The sender then delivers the original application message using the SMSG channel.

When using SMSG channels, but with this dynamic connection approach disabled, this same procedure is used, but is no longer driven by application MPI send requests. Rather, all SMSG channels are setup prior to returning from *MPI_Init*. All mailboxes are allocated upfront. This can use a significant amount of memory for large jobs. Startup time at scale may increase significantly when dynamic connections are disabled.

If the MSGQ facility is used, a *local leader* rank on each node creates a GNI MSGQ. It then exchanges MSGQ channel setup requests with the local leader ranks on the other nodes in the job, using the GNI session management interfaces to exchange any required MSGQ channel setup data. A node–local barrier is performed, and then the other ranks on each node attach to the MSGQ the local leader had previously created and connected.

Table 1. SMSG Maximum Message and Mailbox Size

Job Size	Max. Msg. Size including CH3 hdr	Mailbox Size (bytes) per channel
≤ 1024	1024	4672
> 1024 ≤ 16384	512	2624
> 16384	256	1088

4.3 Eager Message Path

Owing to the relatively short messages that can be delivered by GNI SMSG and especially MSGQ methods, the eager path in the GNI Netmod actually uses two paths. If the application message data and internal MPICH2 CH3 header is under the maximum size message possible for the SMSG mailbox or MSGQ, then the message is delivered using this path alone. The SMSG and MSGQ API's allow for the Netmod to include an internal *tag* with the message. Note this tag has nothing to do with MPI tags used in applications. The tag facilitates handling of packets as the receiver dequeues them from the other end of the SMSG/MSGQ channel. For purposes of this discussion, a tag value of *E0* will be used for the case where the entire message can be delivered using SMSG or MSGQ methods. The use of the tag will be discussed in more detail below.

By default, the maximum size message that can be sent using SMSG varies with the job size, with smaller mailboxes being used as the job size increases (see Table 1). This was done in order to decrease the amount of memory used for SMSG *mailboxes* for larger jobs. The maximum size message – 128 bytes – deliverable using the MSGQ is constant irrespective of job size. With the default settings, the MSGQ facility uses about 74 KB/node for each inter–node connection. Thus for a job spanning 10,000 nodes, about 740 MB is required on each node for the MSGQ.

If the message is larger than can be delivered using GNI SMSG or MSGQ, an RDMA read path is used. The sender process allocates one of the DMA buffers created during the Netmod initialization phase (Section 4.1), and copies the MPICH2 CH3 header and as much of the message data as possible into the buffer. A small control message is then sent through the SMSG/MSGQ channel to the receiver. The message includes the information necessary for the receiver to be able to do a RDMA *read* of the message data from the sender's memory. A different tag value (*E1*) is used to distinguish this message from that used for the path described above for E0 short messages. If there is more message data to be delivered, additional DMA buffers are allocated and the remainder of the message data is copied into these buffers. A small control message is sent for each data buffer used, again with another tag value *E1D*. It is okay for the DMA buffer pool to become depleted. The rest of the message is delivered in correct MPI order as buffers again become available.

On the receive side, when using the SMSG approach, dequeuing of incoming messages is driven by the RX CQ (Section 2.6) associated with the SMSG mailboxes (Section 4.2). The receiver polls the RX CQ to determine which SMSG *mailboxes* have messages to dequeue. The application specific data in the CQE indicates which mailbox to check for messages. In the unlikely event that the RX CQ is *overrun*, the receiver scans all active mailboxes for incoming messages. Processing of incoming messages is driven by the tag value of the message. To maintain correct MPI ordering required by Nemesis, a per–VC pending receive queue was implemented.

When a message with tag E0 is received on the channel, and there are no pending receives, the message is handed off directly to Nemesis using the *MPID_nem_handle_pkt*. There are no *memcpy* calls within the GNI Netmod for this case, although there may be a memcpy to handle unexpected messages within Nemesis itself. If there are any pending receives, then a pending receive structure is allocated off of a free list, and the message is copied out of the SMSG/MSGQ channel and into the pending receive structure. The pending receive is added to the tail of the queue.

When a E1 or E1D tag is received, a pending receive structure and a DMA buffer are allocated. Based on the information in the small control message, either the FMA or BTE is used to initiate a RDMA read of the message data from the send buffer. The pending receive is marked as waiting for completion of the RDMA read and is appended to the tail of the pending receive list. A special DMA buffer is reserved for the pending receive structure at the head of the list to avoid deadlock. In the most recent version of the Netmod, the main pool of DMA buffers is managed using a *buddy* allocator.

As CQEs associated with these RDMA read requests are pulled off the TX CQ, the pending receives are marked as complete. When the CQE for the head of the pending receive list is processed, the message associated with the receive is passed up to Nemesis using the *MPID_nem_handle_pkt*. For a E1 message, the first part of the data is the CH3 header.

A simple ACK protocol is employed in order for the sender to recover DMA buffers after the receiver has completed RDMA reads.

4.4 Rendezvous Message Path

The Nemesis LMT path is used for delivering messages exceeding the eager message size threshold. As described on the Nemesis API wiki [7], the LMT path supports read, write, and cooperative data transfer mechanisms. The uGNI Netmod employs a read method for smaller LMT transfers and a cooperative, RDMA write–based method for longer transfers. The short control messages Nemesis uses for steering an application's MPI messages through the LMT procedure all use the E0 path described above in Section 4.3.

This path utilizes a memory registration cache (Section 4.7). The bandwidth achieved using the LMT path is sensitive to

the efficiency with which the registration cache is being utilized. The efficiency of the RDMA read path is also sensitive to the alignment of the send and receive buffers. Best performance is obtained for this path when the send and receive buffers start at the same relative offset into a cacheline. RDMA writes are much less sensitive to alignment of the send and receive buffers.

4.5 Finalization

Care was taken to fully implement the VC terminate and Netmod finalize callback functions. This was motivated by the near–term need to support checkpoint/restart, and the longer–term desire to support MPI–3 fault tolerance, which essentially entails that an MPI implementation be able to clean up resources associated with failed processes, as well as be able to build new connections with reconstituted processes.

4.6 Network Fault Tolerance

As discussed in Section 2.8, the GNI SMSG and MSGQ facilities guarantee reliable delivery of messages between two EPs. However, GNI does not deal with failed FMA or BTE initiated RDMA transactions. As long as an application requests a TX CQE for each RDMA transaction, the initiator can determine whether or not the transaction succeeded by checking the CQE for errors. A GNI helper function allows the Netmod to distinguish between recoverable CQE errors (e.g. network timeouts) and non–recoverable ones.

The Netmod implements fault tolerance with respect to transient network errors as follows. RDMA transactions are never used directly as a notification mechanism. An *adaptive routing* policy is selected for all RDMA transactions. This helps reduce the number of transactions that may need to be replayed as the result of downed routers. Notification messages go exclusively over the reliable channels made available by SMSG and MSGQ. This allows the initiator of, for example, an FMA RDMA read, to replay the transaction until it succeeds. Notifications, such as the buffer acks for the E1 path, are sent over an SMSG or MSGQ channel once the RDMA transaction succeeds. AMOs are avoided as it is difficult to implement algorithms which allow for recovery when such transactions fail with errors.

The Netmod is only one component of the Cray XE network fault tolerance/fault recovery strategy. Basically the Netmod's role is to ensure that MPICH2 can recover from some dropped messages. Other components include the Hardware Supervisory System (HSS), the Cray XE rerouting software, and the compute node operating system (CLE). A complete description of the mechanism is beyond the scope of this paper.

4.7 uDREG Library and Memory Registration

A registration cache (*uDREG*) was implemented to hide or at least reduce the overhead of memory registration for large message transfers. Since it was known early on in development of the Cray XE software communication stack that other software would also need a registration cache, it was decided to implement the cache as a standalone library. *uDREG* is based on the registration cache implemented in MVAPICH2 [8].

In order to reduce pressure on memory registration resources, the uGNI Netmod does a runtime check for whether or not an application is using *DMAPP* [9], i.e. applications on Cray XE systems using SHMEM, UPC, or CoArray Fortran. If DMAPP is being used by the application, the Netmod doesn't use the UDREG library directly, but invokes DMAPP memory registration functions to register memory regions for LMT transfers. This allows for both DMAPP and the uGNI Netmod to share the same memory registration resources. The uGNI Netmod queries the Gemini NICs to determine the optimal large pagesize to use for SMSG mailboxes and DMA buffers. This also reduces pressure on the NICs registration resources used for registration of 4KB pages.

There are well known pitfalls to using a user–space memory registration cache in the context of the GNU/Linux environment [10]. To avoid the problems cited in [10], a small device driver was developed which utilizes the Linux MMU Notifier facility to inform uDREG when virtual memory (VM) activity by a process has resulted in invalidation of entries in the registration cache. The user–space interface of the device driver described in [10] was retained, although the core of the device driver was completely rewritten and simplified to make use of MMU Notifiers. Note that VM issues attributable to *fork* operations and the Linux Copy–on–Write (COW) feature are handled by kGNI. The application specifies the action taken during the fork operation based on *mode* bits supplied as part of the CDM creation. kGNI makes use of extensions to the MMU Notifier package to handle fork.

5 Basic Performance Characteristics

The intent of this section is to provide basic performance data relevant to the uGNI Netmod and to explain how the data relates to both to the internal operation of the Netmod as well as the Gemini NIC and the Cray XE node architecture. A basic knowledge of the node architecture is assumed in these discussions. For reference, a depiction of the Cray XE node using AMD *Magny Cours* 12–core sockets is shown in Figure 3. All performance results were obtained on a Cray XE with Magny Cours 12–core socket nodes running at 2.0 GHz. The operating system was CLE 3.1.61 and the MPICH2 packaged in MPT 5.3.0.5. Large pages were not used except for one of the bandwidth the tests. Unless explicitly mentioned, default

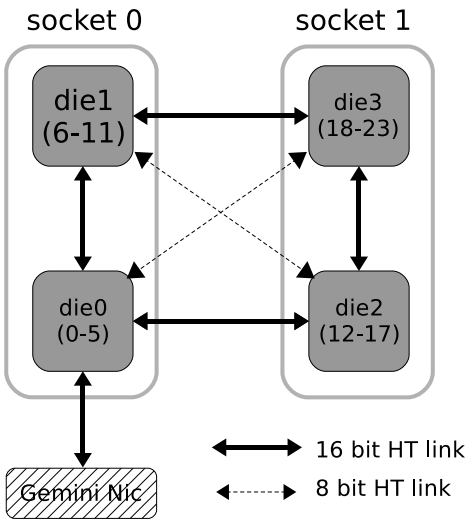

Figure 3. Basic diagram of a Cray XE compute node with AMD Magny-Cours 12 core sockets. A separate memory controller is attached to each die.

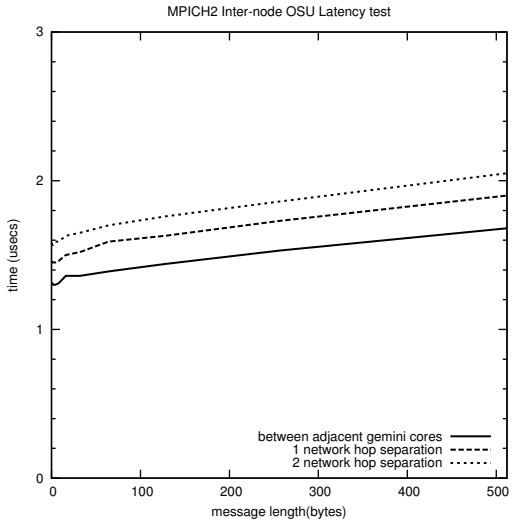

Figure 4. MPICH2 Latency for different network hop counts as measured using the OSU Latency test.

MPICH2 environment variables were used. Results for the MSGQ approach are not included in this paper, as the version of MPICH2 supporting this feature will not be released till later in 2011.

5.1 Message Rate and Latency

The OSU 3.3 MPI latency test was used to measure the latency for MPICH2. Results for various network hop counts are shown in Figure 4. The latency between adjacent Geminis for these test conditions was measured to be a little over 1.3 μsecs. The cost of a network hop for a MPI message is about 150–200 nsecs. The one–way cost of the intra-node hop (not shown on the figure) from one of the cores not adjacent to the Gemini NIC was measured to be about 90 nsecs.

Although the MPI latency for a single sender/receiver pair is useful to know, a more important metric for applications which are typically run using multiple MPI ranks per node is the latency when multiple sender/receivers are trying to exchange messages across a network interface. Figure 5 shows the results from the OSU 3.3 multi-latency (mult_lat) test. The test was run between two adjacent Gemini NICs. To improve the throughput for medium size messages, the MPICH_GNI_RDMA_THRESHOLD environment variable was set to 16384 for this test. The MPICH_GNI_MBOX_PLACEMENT environment variable was set to specify *nic* placement for the SMSG mailboxes and CQs. This results in the GNI Netmod placing the SMSG mailboxes and CQs on the memory of die0 (see Figure 3). Owing to the way the coherent HyperTransport protocol handles upstream traffic from an I/O device into the node, this gives much better performance than if the mailboxes and CQs are placed local to the MPI ranks. One observes that very good latency is observed for small messages even when there are 24 ranks/node up to 1024 bytes. It is at this point that the MPICH2 switches to the E1 protocol and the DMA buffers begin to be used. At the largest message lengths shown in the figure, the latency is beginning to be dominated by the serializing effect of the BTE. This effect should be diminished in the next major release of CLE, in which multiple channels of the BTE will be available to applications.

The aggregate message rate for short and medium size MPI messages is shown in Figure 6. These measurements were made also made with the MPICH_GNI_MBOX_PLACEMENT environment variable set to specify *nic* placement. The MPICH_GNI_RDMA_THRESHOLD environment variable was not set for these measurements. The maximum message rate realized with this placement option, and using 2.0 GHz processors, is about 8 million MPI messages/sec. Rates over 9 million messages per second are measured with faster processors. The drop off in message rate at 1024 bytes is due to the switch from the E0 to the E1 protocol (Section 4.3).

5.2 Bandwidth

Bandwidth measurements were made using the IMB 3.2.2 PingPong test and various OSU 3.3 bandwidth tests. Unless otherwise mentioned, all tests were run between adjacent Gemini NICs.

Results of the IMB PingPong test are shown in Figure 7 for various ways of handling large messages. As shown in the figure, the best bandwidth is obtained when using the LMT path described in Section 4.4 and also using *lazy* memory deregistration for the registration cache. Lazy memory deregistra-

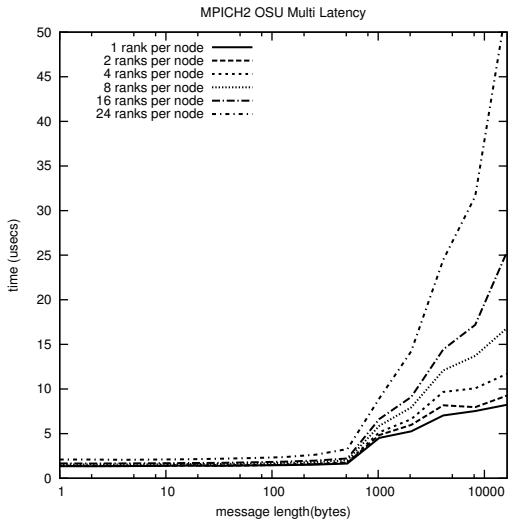

Figure 5. MPICH2 Latency for multiple sender/receiver pairs per node.

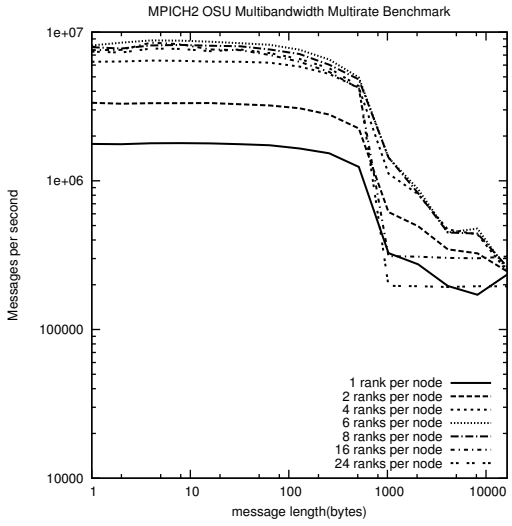

Figure 6. MPICH2 message rate measured using the OSU mbw_mr test with different numbers of MPI ranks per node.

tion is the default policy used by MPICH2. The bandwidth drops significantly if the lazy memory registration policy is not used. Not using the LMT path at all has a similar effect on the bandwidth for large messages. The drop in bandwidth between 512 and 1024 bytes is again due to the switch from the E0 to E1 protocol in the eager path. The differences in bandwidth for the longer transfers methods only appear at 8 KB and above because that is the default threshold for switching from the eager to the rendezvous protocol.

Since many MPI applications are typically run with multiple processes per node, bandwidth results when using multiple MPI send/receive pairs are shown in Figure 8. For this test, the MPICH_GNI_RDMA_THRESHOLD environment variable was again set to 16384. The bandwidths are derived from the latencies obtained using the OSU 3.3 multi_lat test. These are the results in bandwidth rather than latency, for messages longer than those shown in Figure 5. At transfer sizes beyond 16384 bytes, the available bandwidth per rank is dominated by the effects of sharing the BTE between the ranks for transferring the message data. The reason for the dip at 512 KB and 1 MB transfer lengths for the single rank per node case is under investigation.

Figure 9 is included to show effects of the MPICH2 Nemesis design on the bandwidth realized using different MPI methods for transferring data, and also to show results of the OSU bidirectional bandwidth test. As discussed in Section 4.4, the Nemesis device currently does not use the LMT path for MPI-2 RMA transfers. Thus, the realized bandwidth for MPI_Put and MPI_Get operations is similar to that obtained for long MPI_Send messages when the LMT path is disabled (Figure 7). Again the dip between 512 and 1000 byte transfers arises from the E0 to E1 transition. For the osu_bw test using base 4KB pages for the send and receive buffers, a bandwidth of around 3.8 GB/sec was measured, at 16384 byte message sizes, and 4.4 GB/sec for 1MB message sizes. The decreasing bandwidth above 1MB for the MPI_Put and MPI_Get probably is due to caching effects associated with the buffered transfer protocol used. The figure also includes results when using large pages for the osu_bw test. Large pages give much better performance for longer messages, approaching the performance obtained using DMAPP. Modifying the osu_bw test to test larger message sizes, an asymptotic bandwidth of 6 GB/sec is realized for very large messages (64 MB) when using large pages.

6 Future Work

One of the main areas of focus for enhancement of the uGNI Netmod is providing better support for independent progress of the state–engine, and hence allowing for opportunities for better overlap of computation with communication. Currently, kGNI provides some level of support for offloading to the Gemini BTE by allowing applications to queue BTE transfer requests in the kernel. As the BTE processes trans-

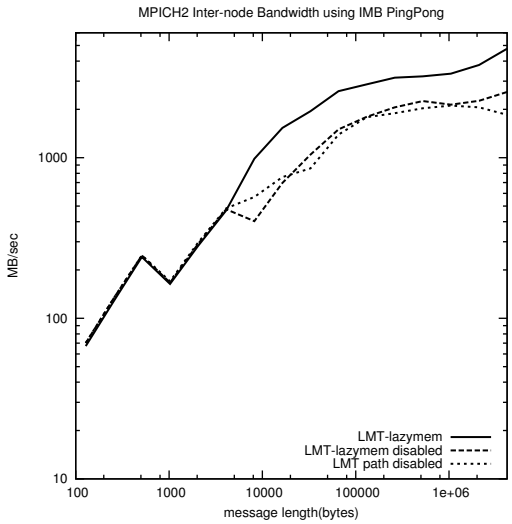

Figure 7. MPICH2 Inter–node IMB PingPong Bandwidth using various options for handling long messages.

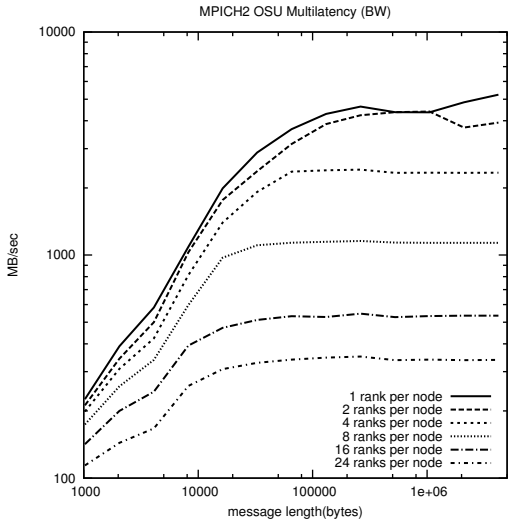

Figure 8. MPICH2 bandwidth per rank for multiple ranks per node as derived from the latency measurements obtained using the OSU multi_lat test.

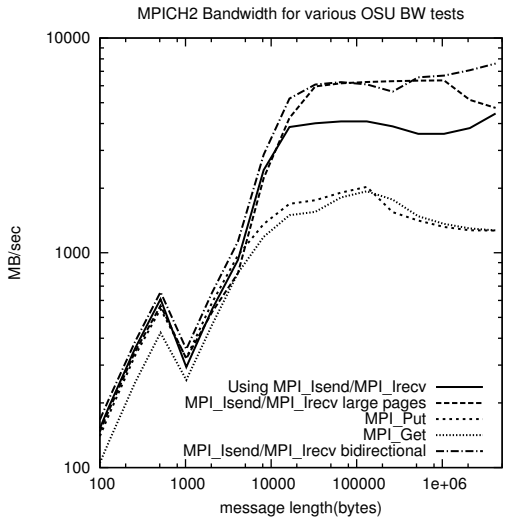

Figure 9. Comparison of realized bandwidth for MPI_Send/MPI_Recv and MPI_Put and MPI_Get. Also shown is bidirectional bandwidth obtained using the OSU bibw test.

fer requests, kGNI is able to enqueue more requests into the BTE's hardware request queues without the application having to make MPI calls. The uGNI Netmod will need to be enhanced to leverage this support. Approaches being investigated include enhancing of the existing asynchronous–thread infrastructure within MPICH2, as well as more complex approaches (e.g. [5]) which make use of the core–specialization features available in CLE.

Longer–term, work on the Netmod will include adding support for MPI-3 features such Fault Tolerance and extended MPI-3 RMA functionality.

7 Acknowledgments

The authors would like to thank Steve Oyanagi (Cray) for collecting much of the data presented in this paper. The authors would also like to acknowledge Kim McMahon (Cray) for enhancing the Cray PMI library to support the KVS functionality required by Nemesis.

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

References

[1] Robert Alverson, Duncan Roweth, and Larry Kaplan. The Gemini System Interconnect. *High-Performance*

Interconnects, Symposium on, 0:83–87, 2010.

[2] Darius Buntinas, Guillaume Mercier, and William Gropp. Design and Evaluation of Nemesis, a Scalable, Low-Latency, Message-Passing Communication Subsystem. In *CCGRID'06*, pages 521–530, 2006.

[3] Fault Tolerance Working Group. Run-though Stabilization Interfaces and Semantics. `svn.mpi-forum.org/trac/mpi-forum-web/wiki/ft/run_through_stabilization`.

[4] Michael Karo, Richard Lagerstrom, Marlys Kohnke, and Carl Albing. Application Level Placement Scheduler (ALPS). In *Proceedings of Cray User Group 2006*, 2006.

[5] Ping Lai, Pavan Balaji, Rajeev Thakur, and Dhabaleswar K. Panda. ProOnE: a General-purpose Protocol Onload Engine for Multi- and Many-core Architectures. *Computer Science - R&D*, pages 133–142, 2009.

[6] MPICH2. `www.mcs.anl.gov/research/projects/mpich2/`.

[7] MPICH2–Nemesis. Nemesis Network Module API. `wiki.mcs.anl.gov/mpich2/index.php/Nemesis_Network_Module_API`.

[8] Network–Based Computing Laboratory. MVAPICH: MPI over Infiniband, 10GigE/iWARP and RoCE. `mvapich.cse.ohio-state.edu/overview/mvapich2`.

[9] Monika ten Bruggencate and Duncan Roweth. DMAPP–an API for One–sided Program Models on Baker Systems. In *Proceedings of Cray User Group 2010*, 2010.

[10] Pete Wyckoff and Jiesheng Wu. Memory Registration Caching Correctness. In *Proceedings of CCGrid05*. IEEE Computer Society, 2005.