Title

An Evaluation of One-Sided and Two-Sided Communication Paradigms on Relaxed-Ordering Interconnect

ISBN

978-1-4799-3799-8

Authors

Ibrahim, Khaled Z

Hargrove, Paul H

Costin, Iancu

et al.

Publication Date

2014-05-01

DOI

10.1109/ipdps.2014.116

Peer reviewed

An Evaluation of One-Sided and Two-Sided Communication Paradigms on Relaxed-Ordering Interconnect

Khaled Z. Ibrahim, Paul H. Hargrove, Costin Iancu, and Katherine Yelick
Lawrence Berkeley National Laboratory, Berkeley, USA
Email: {kzibrahim, phhargrove, cciancu, kayelick}@lbl.gov

***Abstract*—The Cray Gemini interconnect hardware provides multiple transfer mechanisms and out-of-order message delivery to improve communication throughput. In this paper we quantify the performance of one-sided and two-sided communication paradigms with respect to: 1) the optimal available hardware transfer mechanism; 2) message ordering constraints; 3) per node and per core message concurrency. In addition to using Cray native communication APIs, we use UPC and MPI micro-benchmarks to capture one- and two-sided semantics respectively. Our results indicate that relaxing the message delivery order can improve performance up to 4.6× when compared with strict ordering. When hardware allows it, high-level one-sided programming models can already take advantage of message reordering. Enforcing the ordering semantics of two-sided communication comes with a performance penalty. Furthermore, it seems that exposing out-of-order delivery at the application level is required for the next generation of programming models. Any ordering constraints in the language specifications reduce communication performance for small messages and increase the number of active cores required for peak throughput.**

I. Introduction

Hardware vendors traditionally employed a combination of Remote Direct Memory Access (RDMA) and out-of-order packet delivery to provide communication throughput on large scale multicore systems. At the system level API (Application Programming Interface), messages were usually ordered to meet the semantic requirements of higher level abstractions. Recently, the Cray Gemini hardware and APIs started exposing multiple message ordering modes to its clients. The main contribution of this work is the evaluation of how well equipped to take advantage of hardware out-of-order message delivery are existing one-sided and two-sided programming models or communication libraries. To our knowledge, ours is the first study to examine in detail the usage of this functionality in implementations of programming model abstractions.

The predominant paradigm for the last twenty years has been the Message Passing Interface (MPI) with its two-sided synchronization semantics. The non-blocking Isend/IRecv communication primitives in MPI can internally take advantage of un-ordered messaging.

One-sided communication has started to gain popularity roughly ten years ago, as showcased by the Unified Parallel C [1] programming language. Until recently, the UPC language standard provided only blocking communication and a relaxed order memory model to allow compiler or runtime optimizations. In Nov 2012, both the UPC 1.3 (draft) [2] and the MPI 3.0 specifications [3] introduced user level non-blocking one-sided communication primitives and brought to the forefront the question of the efficacy of existing hardware support for such operations.

RDMA is usually supported by a variety of hardware and software mechanisms targeting short, medium and large transfers, while the message ordering is enforced by a cooperation of node and network hardware protocols, e.g. memory or PCI controller and Network Interface Card (NIC). The lowest level Application Programming Interface (API) exposes fine grained control over hardware functionality and runtime implementors face multiple challenges: 1) choosing the optimal hardware transfer mechanism; 2) enforcing the message ordering required by the high level protocol using the system level mechanisms and; 3) handling memory registration.

Our results indicate that out-of-order message delivery improves native communication performance by as much as 4×. The one-sided paradigm is able to exploit very well the available hardware support and provides for any given transfer the best attainable performance. When the target memory region is not registered, either usage of bounce buffers or dynamic memory registration is required in the runtime implementation. In this case we observe as much as 10% performance degradation for small messages with bounce buffers and as much as 7% performance degradation for large messages with dynamic registration.

The MPI implementation requires message ordering for message matching, while the actual data transfer may be performed without ordering. Overall, the MPI implementations examined can lose 30% of the peak sustained bandwidth, even for large messages. The performance loss is caused by a combination of not using the best available transfer protocol, protocol overhead and memory registration. Our analysis indicates that the impact of protocol overhead, bounce buffers and dynamic registration accounts only for 5%-10% performance loss, while the inability to exploit out-of-order message delivery on Gemini accounts for 23%-30% performance loss.

The rest of the paper is organized as follows. We introduce the study platform and methodology in § II. In § III, we analyze the performance of the Cray Gemini native communication APIs DMAPP and GNI with respect to message ordering. This provides us with upper bounds for the performance attainable by the implementation of high level language or communication runtimes. We present semantic and performance analysis of the of UPC one-sided communication in § IV and MPI two-sided communication in § V. We present related work in § VII and conclude in § VIII.

II. Experimental Platform

The system used for this evaluation is a Cray XE6 installed at the National Energy Research Scientific Computing Center (NERSC). As shown in Figure 1, each node contains two 12-core AMD MagnyCours 2.1-GHz processors and 32GB of DDR3 memory. Cores are grouped into four NUMA domains and communicate using HyperTransport 3.0 [4]. The nodes are connected using Cray's Gemini router ASIC and a 3-D torus network topology. Two nodes are connected to a Gemini NIC through HyperTransport. Each Gemini chip has 48 ports, eight internal and 40 external arranged in ten network connections, two each in +X, -X, +Z, -Z, and one +Y and -Y . Each of the ten Gemini torus connections is comprised of 12 lanes with aggregate bandwidths of 4.68 to 9.375 GBytes/sec per direction. To support runtime and compiler developers, Cray provides multiple hardware messaging mechanisms.

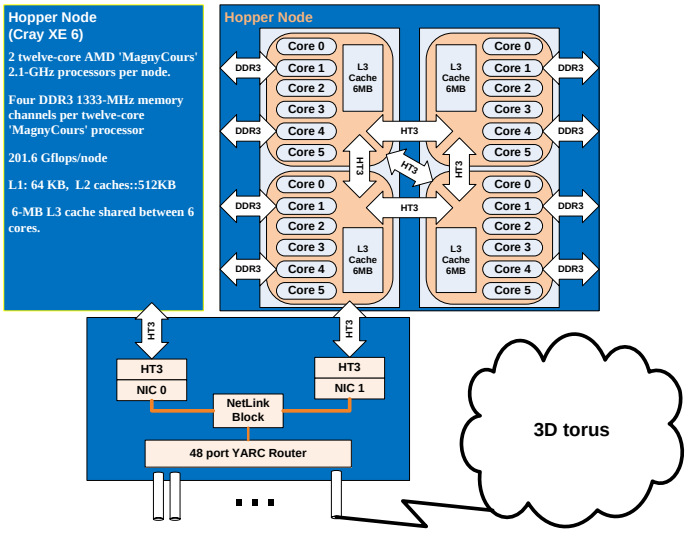

Figure 1. Cray XE6 node architecture.

Hardware Transport: Cray exposes two hardware mechanisms for remote memory access: Fast Memory Access (FMA) and Block Transfer Engine (BTE). FMA is intended for efficient transfer of small messages and uses a set of memory windows to allow direct data movement of the user-space data through the ASIC. FMA supports different types of transactions including short message (SMSG), Shared Message Queue (MSGQ), FMA DM for Get and Put, and atomic memory operations (AMO). BTE is implemented on the ASIC and once started, any communication is completely managed by the hardware. The cost of starting up this mechanism is high, making it more suitable for large memory transfers. On Gemini, only 4 BTE requests can be active per node concurrently.

Message Ordering: The Cray Gemini interconnect provides multiple ordering modes for outstanding transactions for the FMA and BTE mechanisms. As the Gemini chip uses HyperTransport to communicate with the memory and CPUs within a node, these modes reflect both HT and Gemini capabilities. HT supports *posted* and *non-posted* requests. Posted requests do not expect a response for completion, *e.g.* memory writes and messages. Non-posted requests expect responses, such as memory reads and write operations that request acknowledgement.

At the Gemini level, three message ordering modes are exposed by the API. With *strict* ordering, all local memory and network operations are completed in the order they are issued by the processors or by the Gemini chip, without overtaking. The second mode is the *default* setting for the Cray APIs and it allows for ordering where non-posted requests (Gemini-level Gets in particular) may pass posted writes. The third mode, labeled *relaxed* ordering in Gemini, allows for reordering of non-posted and posted requests in addition to the passing allowed by the *default* setting. As shown later, the three modes can have a significant impact on the observed performance.

System APIs: On Gemini, the Generic Network Interface (GNI) exposes FMA and BTE control, message ordering and support for implementing Active Messages [5]. GNI is thus well suited for runtime implementations of one-sided or two-sided communication paradigms. The Distributed Shared Memory Application interface (DMAPP) supports Partitioned Global Address Space languages using one-sided communication, and also provides collective operations. DMAPP hides the FMA and BTE support and selects internally the best mechanism for a transfer. The Cray UPC compiler is implemented using DMAPP. The follow-on Cray Aries [6] network also supports both GNI and DMAPP.

A. Experimental Methodology

To study communication performance we extend the OSU_MBW_MR benchmark [7], [8] to support bidirectional bandwidth measurements[1], as shown in Figure 2. To understand the attainable native performance, we implement GNI FMA, GNI BTE and DMAPP versions and experiment with message ordering. We use the Cray Programming Environment 4.1.40. For UPC one-sided communication we use a version implementing the new non-blocking communication support introduced in the UPC 1.3 (draft) [2] language specification. We have evaluated both Cray and Berkeley UPC.

The microbenchmark in Figure 2 issues a `window` of non-blocking messages before checking for their completions. For single-sided communication, we post the memory requests depending on the underlying library: GNI_postFma for FMA, GNI_postRdma for BTE, dmap_get_nb/dmapp_put_nb for DMAPP, upc_memput_nb/upc_memget_nb for Cray UPC[2]. We then use a corresponding wait for completion based on the posted requests. For two-sided communication, we post non-blocking receives before posting non-blocking sends. We then wait for completion of all posted requests. We made sure all timing measurements are consistent across all microbenchmark implementations.

[1]For one-sided version of the microbechmark, no buffer is reused within a window of request. This is not requirement for the two-sided version.

[2]Berkeley UPC provides similar non-blocking interfaces bupc_memget_async and bupc_memput_async

Most of the experiments done on this paper use a 4-node configuration, *i.e.* up to 96 cores.

```
1: procedure POINT_TO_POINT_MESSAGES(s)
2:     for i = 1 → window_size do
3:         issue a non-blocking receive request with buffer size s
4:     end for
5:     for i = 1 → window_size do
6:         issue a non-blocking send request with a buffer size s
7:     end for
8:     Wait all requests completion
9: end procedure
10:
11: procedure SINGLE_SIDED_MESSAGES(s)
12:     for i = 1 → window_size do
13:         Create a request of size s
14:         Post Get/Put between local and remote memories
15:     end for
16:     Wait all requests completion
17: end procedure
18:
19: function MAIN
20:     num_pairs ← ranks/2
21:     target ← (my_rank + num_pairs)%ranks
22:     for s = 1 → msg_size_count do
23:         size ← msg_size[s]
24:         for i = 1 → repeat + skip do
25:             if i = skip then
26:                 stime = time()
27:                 Barrier
28:             end if
29:             Call POINT_TO_POINT_MESSAGES(size)
30:             or SINGLE_SIDED_MESSAGES(size)
31:         end for
32:         Barrier                    ▷ or calculate max transfer time
33:         etime ← time()
34:         if my_rank = 0 then
35:             bandwidth[s] ← size * loop * ranks/(etime − stime)
36:         end if
37:     end for
38: end function
```

Figure 2. Microbenchmark for measuring bidirectional bandwidth on two-sided and single-sided communication paradigms.

III. Analysis of Low-level Transport Protocols on the Cray XE6 System

Figure 3 shows the performance of different message ordering modes for DMAPP, FMA, and BTE. We plot the aggregate bandwidth for 48 pairs of communicating cores[3], each having one outstanding get transfer at any given time. For brevity we present data only relaxed ordering. Default ordering yields performance never better than relaxed or worse than strict, and is generally closer to strict. The put operations shows similar performance trends in responding to ordering relaxation. Surprisingly, the peak performance of put operations is about 20% lower than the get operations.

As expected, FMA provides the best performance for small messages and BTE is best for messages larger than 4K. DMAPP hides the hardware complexity and follows the same trends as FMA for small messages and as BTE for large messages by automatically switching protocol based on the message size. However, DMAPP also allows the library user to control the switching between the two protocols if desired. Generally, DMAPP achieves the same performance as a well tuned GNI based implementation.

[3]We used four hopper nodes to make traffic traverse the network. Two-node traffic goes through the NIC bypass.

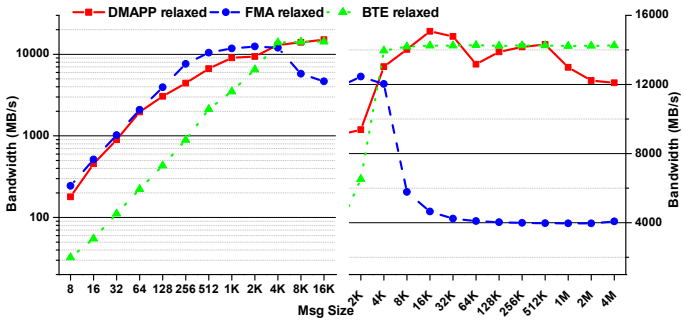

Figure 3. Performance of DMAPP and low-level GNI communication protocols for relaxed ordering. Forty-eight communication pairs, each issuing a single get operation.

Switching between the FMA and BTE protocols has a typical default value of 4KB on most runtimes. The best switching point typically depends on multiple factors including the level of concurrency within the node, the number of outstanding requests per processor, and the size of the job (number of nodes).

A. Concurrent Communication and Ordering

The impact of relaxed ordering becomes more apparent when examining bandwidth in conjunction with the number of active cores per node. This is typically the case when applications use multiple ranks per node for MPI, or multiple threads for UPC.

As shown in Figure 4(right), the best performance of BTE with strict ordering is obtained when using only four cores per node. We typically expect that increased concurrency should improve performance by mitigating the cost of transfer setup. Surprisingly, not only does the bandwidth saturate before the number of threads is equal to the 24 cores on each node, but substantial slowdowns of 2-5× are observed when using too much concurrency. This has important implication for flat (non-hybrid) programming models, either MPI or UPC, which would have at least one thread per core communicating with the network. While one thread per core may be sub-optimal for other reasons having to do with the local memory system, we show in this section that it also has a negative effect on communication performance with strict ordering.

The aggregate bandwidth when using 24 cores per node and strict ordering is as much as 4.6× lower than the best attainable bandwidth using BTE, which occurs when using just 4 cores per node. As will be shown next, this drop-off does not occur with relaxed ordering. Therefore, we conjecture that because strict ordering at the HT level cannot distinguish transfers from independent ranks, the increased concurrently is producing interference rather than providing an opportunity for overlapping independent Get operations.

For BTE with relaxed ordering, shown in Figure 5(right), the performance improves monotonically with the increase

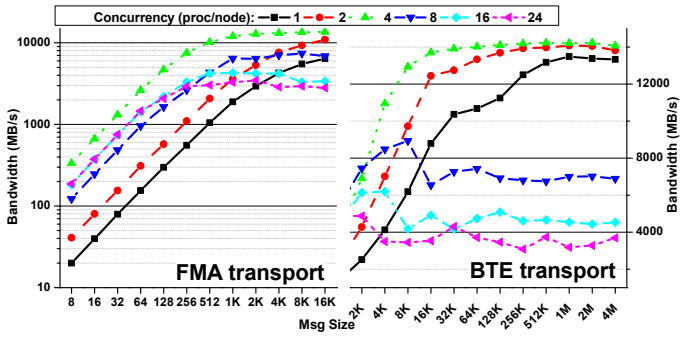

Figure 4. Interaction between *strict* ordering and concurrency for FMA and BTE get operation (48 pairs).

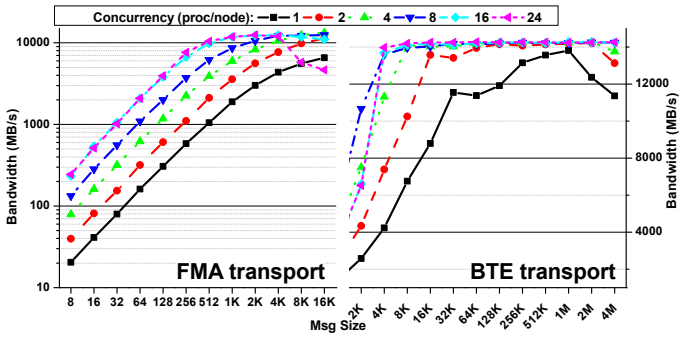

Figure 5. Interaction between *relaxed* ordering and concurrency for FMA and BTE get operation (48 pairs).

in the number of active cores, until it saturates. Moreover, for medium size messages (between 4KB and 16KB) the performance with low concurrency is better than that with strict ordering, by up to 20%.

For FMA, increasing the concurrency with strict ordering also hurts the performance. As shown in Figure 4(left), the best bandwidth is achieved with four cores per node for small and large messages. With relaxed ordering, shown in Figure 5(left), the performance improves with increased concurrency of injection. This trends continue for messages less than a page size (4KB). For large messages, FMA is significantly slowed down with concurrency, even with relaxed ordering due to exhausting the FMA resources, not shown in Figure. In fact, we are surprised that FMA can even deliver a large percentage of the peak sustained bandwidth under low concurrency for large messages.

The presented data show the importance of exploiting relaxed ordering by the high-level programming paradigms, whether it is one- or two-sided.

B. Concurrency and Memory Registration

Memory registration is typically needed when offloading communication to the RDMA interconnect hardware. For the Get operations used in our study, the source and destination message buffers must be known to the interconnect (registered), regardless of the choice of message ordering. All experiments shown earlier were done using buffers pre-registered prior to the timing loops, thus the costs of registration were not considered.

Registration of the memory with the NIC prevents the physical memory from being swapped by the OS, and enables the NIC to access the memory without interrupting the CPU. Memory registration resources are typically limited shared resources. On Gemini NIC the number of pages that can be registered is typically fixed among all running processes (or threads). If 4K page size is used, only a fraction of the total physical memory can be registered (<800MB) per node.

As shown in Figure 6, the memory registration cost typically increases with the number of concurrent requests as it requires serialization of access to the shared registration resources. For instance, registering a single 4KB page(or less) takes about 0.9 μsec. With 24 concurrent requests, the same operation takes about 30 μsec. Memory de-registration latency shows similar trends in performance in terms of the cost of operation and being affected by concurrency.

Handling memory registration is an important design choice in the implementations of high-level communication runtimes. Typical choices include the maximum registration segment size, the page granularity, the use of registration cache, etc. Concurrent registration has a negative impact on performance independent from message ordering relaxation. Most runtimes try to remove registration operations from the critical path of execution.

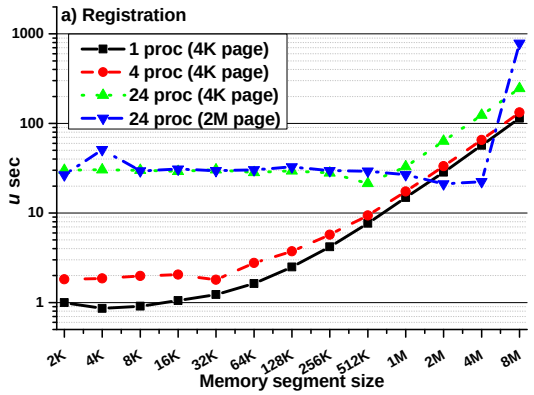

Figure 6. Memory registration overhead for different segment granularities and page sizes (μsec=2100 cycles).

C. High-Level Abstractions

When implementing higher level communication protocols, developers have to choose the best performing hardware transport mechanism. On Cray Gemini, this translates into choice of DMAPP, FMA or BTE and how to handle registration. In all cases the developer also has to select a message ordering that allows for efficient implementation of the semantics of the higher level protocol. To illustrate these choices we choose two representative communication paradigms, one-sided as illustrated by UPC and two-sided as illustrated by MPI.

MPI represents the de-facto standard for distributed memory programming and it exposes a two-sided protocol. MPI provides for non-blocking communication and not fully specified message orderings, but it also provides deterministic execution guarantees in most cases.

In the rest of this study we focus on the the usage of ordering relaxation with both one-sided and two-sided

communication models. It is not our intention to perform a full performance comparison between MPI and UPC. We aim to identify parts of their respective specifications that enable or preclude optimizations. We use UPC as the proxy for one-sided, primarily because its implementations attain performance identical to native. Meanwhile, MPI's send/receive operations are used to represent two-sided communication.

We have also performed the experiments described in this paper with MPI-2 one-sided, using Cray and Open MPI implementations, and observed lower performance than MPI two-sided, roughly 10% to 25% of the two-sided performance. As the one-sided features in MPI-3.0 become more widely available and fully optimized, we expect their performance to be similar to UPC or GASNet.

IV. Analysis of UPC One-Sided Communication

UPC provides an abstraction of Partitioned Global Address Space (PGAS) models that use one-sided communication primitives. Its memory model exposes strict and relaxed memory orderings, similar to the GNI level notions. With strict ordering, operations issued by one thread are observed in order, while relaxed allows message reordering. A compiler or runtime can exploit reordering to improve performance only when single rank dependencies cannot be violated and no aliasing ambiguity obstructs such dependency checks.

A. One-sided Communication and Transport Ordering

One-sided communication in the PGAS models views distributed memory as a natural extension to the memory. As such, operations targeting the memory of the issuing thread are mapped to loads/stores while those targeting memory of other threads are mapped to Get/Put operations (though some implementations may use loads/stores to access all memory within the same compute node). In UPC, the ordering of these operations can be either strict or relaxed as specified by the program and the runtime needs to provide consistency across intra- and inter-node memory accesses. Relaxed ordering allows the compiler to reorder memory operations to improve performance. It also allows the compiler to know when intra-node memory fences are needed to guarantee the ordering intended by the application developer. Intra-node ordering is enforced using memory fences, while system networking APIs usually guarantee strict ordering of transfers and with respect to in-node operations. To our knowledge control over relaxed versus strict ordering of Get/Put operations is available only on Gemini (and it's successor, Aries), which opens new possibilities for improving performance.

Relaxed ordering violates the intuitive "sequential consistency" (SC) model because operations executed by one process may appear out-of-order by another. Figure 7 shows an example of SC violation. The process (P_0) Gets a remote datum and then Puts a new value to the same location. Then P_0 informs process P_1, on the same node, of this update. If P_1 Gets this remote datum while relaxed ordering is enabled, the resulting non-posted read may pass the posted write of the Put by P_0. This reordering allows a race in which P_1 retrieves a stale datum from the remote node[4].

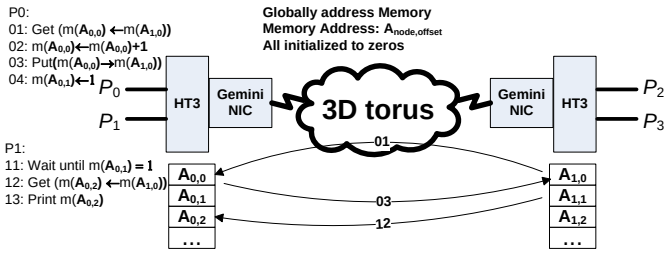

Figure 7. A UPC ordering challenge on relaxed ordering interconnect. Can the interconnect reorder operations 03 and 12?

This example illustrates the interplay between the message ordering models and the completion semantics of one-sided communication operations. Local completion semantics ensure that it is safe to reuse the initiator's buffer. Strict ordering by hardware typically requires two conditions: program order in issuing requests and write (or Put) atomicity. In UPC, program order is typically guaranteed by the compiler when variables are declared strict. Atomicity of write (or Put) can be either achieved by global visibility by ensuring the operation is committed at the destination, or global completion by ensuring that the Put operation is received by a serialization agent at the receiver. This agent is responsible for creating a global order. Library based approaches such as MPI-3 provide local completion as default while global completion requires use of explicit synchronization primitives at the application level. While GASNet and the UPC language provide the option of relaxed semantic at the application level and exploit that to improve performance [9], for operations on strict UPC variables they provide global completion semantics. On Gemini, the hardware provides global completion by default when enabling the strict message order, but this is expensive as discussed in Section III-A. When enabling relaxed message ordering, the GNI interface provides only local completion semantics for Put operations and the runtime needs to perform additional work to make sure that message is delivered to the destination. Unlike GNI, this completion semantic issue is handled transparently by the DMAPP interface.

B. One-Sided Communication and Memory Registration

In MPI-2 one-sided, regions of remotely accessible memory are identified only at runtime through window creation operations, and memory registration must be performed dynamically. However, most one-sided programming models either make explicit which memory may be accessed remotely (as with the `shared` type qualifier in UPC) or are amenable to compiler analysis to make this identification. This can enable the allocation of remotely accessible objects in regions of memory which are registered statically, at program start-up.

[4]This kind of race condition prevented GASNet from adopting relaxed ordering in earlier releases

Only when the local buffer does not reside in pre-registered memory does dynamic registration become a concern in a one-sided model. For small transfers a typical implementation uses pre-registered "bounce buffers"[5], and performs dynamic registration only for large transfers.

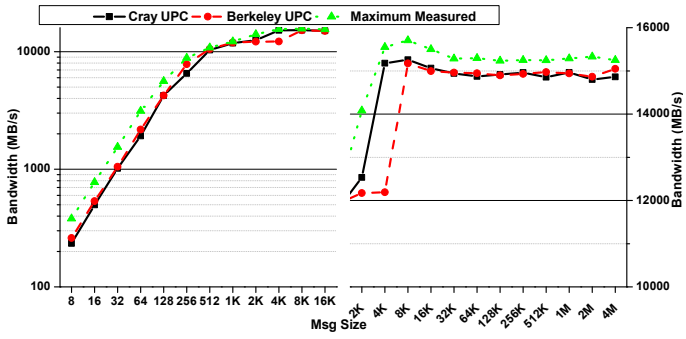

Figure 8. Cray and Berkeley UPC performance implemented on DMAPP and GNI interfaces respectively. Performance is compared with best observed performance on each message size.

C. Performance Analysis of One-sided Communication

In Figure 8 we present the micro-benchmark performance for the Cray and Berkeley UPC runtimes. The "Maximum Measured" series reports the best performance attained by any combination of FMA-vs-BTE, number of cores active, and message ordering. However, in all cases the maximum performance comes from relaxed ordering. Cray UPC is implemented on top of DMAPP, while Berkeley UPC is implemented on top of GASNet which uses GNI and switches between FMA and BTE based on transfer size. Thus we compare DMAPP with hand tuned GNI. For brevity, we include results for 24 active cores and window size of 1.

Comparing the BUPC with CUPC performance illustrates that DMAPP is successful at hiding the machine complexity as it produces performance comparable with tuned GNI code. The difference relative to the maximum is typically largest for small messages, because of the ability of the interconnect to carry more messages. While local completion can reduce the perceived latency of a Put operation at the sender, it does not affect the throughput of the network for Get operations. As such, we have not noticed in our Get benchmarks any sizable degradation in throughput for using global completion instead of local completion. This also shows that the observed performance of a communication library on GNI or DMAPP should match for the same communication semantics. Any difference should be attributed to either tuning (or exploitation) of the underlying primitive or the communication paradigm's constraints and semantics.

For the rest of this paper, we limit our discussion to Cray UPC, while noting that similar trends are observed by Berkeley UPC. Thus, contrasted results for UPC one-sided and MPI two-sided implementations are from the same vendor, Cray.

In Figure 9, we show the variation of the bandwidth with concurrency and window size for four message sizes. These

[5]FMA put operations are an exception: they do not require registration of the source buffer.

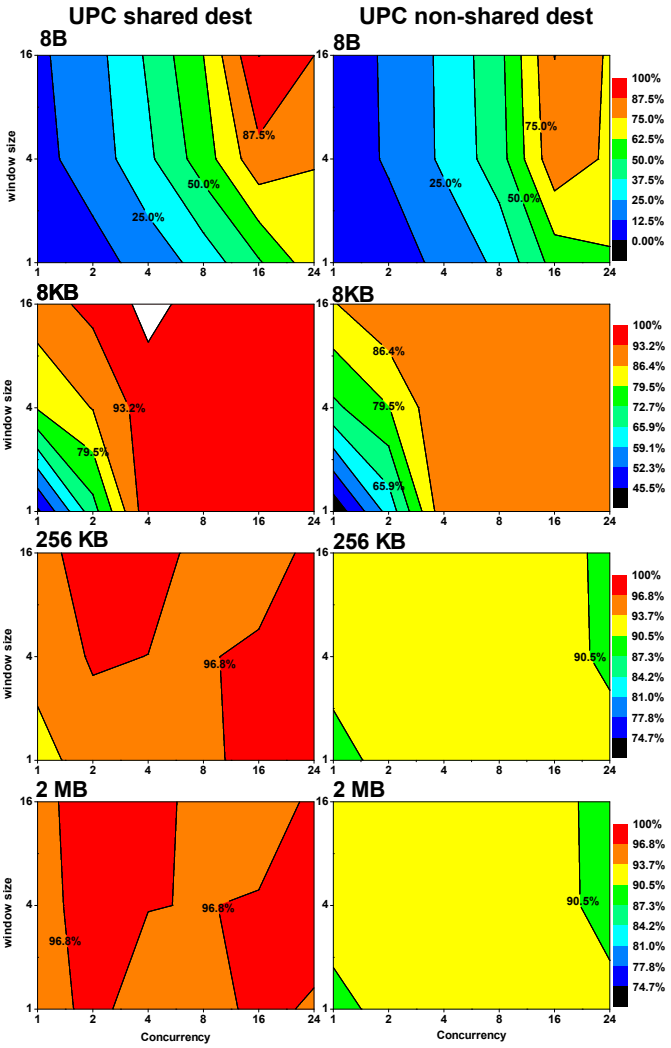

Figure 9. Concurrency *vs.* window size impact on bandwidth performance (MB/s) of single-sided UPC get for shared *vs.* non-shared data.

four sizes correspond to those needed to sample the four protocols that Cray's MPICH-based MPI uses for message transfers, as explained in Section V. We present results for a target of a Get operation either in the registered memory space (shared) or not (non-shared). For small messages, the aggregate bandwidth increases with the number of messages in flight, regardless of whether the increase comes from window size or core count. As FMA exhibits a large injection overhead, performance benefits more from concurrent injection and increasing the number of cores active per node.

Comparing between the shared and non-shared data cases, the behavior for small transfers does not differ much because the typical practice of copying via bounce-buffers has a relatively small overhead compared to the network latency. For large transfers, operations on shared destination memory are nearly all within 5% of the peak sustained bandwidth and little variation is seen with varied concurrency or window size. For non-shared destination, however, we lose about 7% of the performance versus transfers to shared regions. For 256KB and 2MB operations, we notice that only non-shared

performance degrades at high concurrency and large outstanding requests per core, which we attribute to the memory registration overhead which increases with the concurrency level, shown in Figure 6 and discussed in Section III-B.

V. Analysis of Two-Sided Communication

In order to provide the semantic guarantees prescribed by the standard, MPI implementations are subject to several practical constraints. The Send/Recv matching order and message cancellation semantics restrict the choice of hardware message ordering. Support for message probing and message size ambiguity at the receiver side can delay the choice of a transfer protocol and any corresponding memory registration or buffer allocation.

A. Ordering Constraints in MPI

The two-sided ordering constraints in MPI apply mostly to operations issued by a given origin process to the same target process. The goal of the specification is to ensure platform-independent deterministic program execution between pairs of ranks even if the programmer does not fully constrain message matching, for instance by the use of unique tags.

B. MPI Message Matching

The MPI standard includes the following non-overtake rule [3] that facilitates matching of sends to receives. *A*) If a sender sends two messages (Message 1 and Message 2) in succession to the same destination, and both match the same receive, the receive operation will receive Message 1 before Message 2. *B*) If a receiver posts two receives (Receive 1 and Receive 2), in succession, and both are looking for the same message, Receive 1 will receive the message before Receive 2. MPI guarantees that message-passing code is deterministic between a pair of processes, if processes are single-threaded, even if wildcard MPI_ANY_SOURCE is used in receives [3]. The same ordering guarantees should be in place if the same tag is reused for successive messages from the same source to the same destination. This means that ordering is required in message matching, requiring either use of strictly ordered delivery or a mechanism (such as sequence numbers) to reconstruct the sender's order. However, the progress of the transfers can be relaxed, if runtime can satisfy all other constraints.

Figure 10 shows an example where a process P1 issues two MPI_Isend operations to P2, which uses wildcard receives. In this case the messages must be matched in order, but the data transfers can proceed concurrently using relaxed order, without care about possible race conditions. Moreover, the two non-blocking receives posted by P4 need to processed by the order of posting the receives not the order of message arrivals. As such processing the message P2→P4 cannot proceed before the arrival of the message P0→P4.

We have examined two MPI implementations for Gemini. Cray MPI implements message matching using strictly ordered mailboxes (or FMA) to guarantee ordering constraints. Open MPI [10] typically uses ID sequence numbers for matching, but this translates at the RMA level again into using mailboxes and strict ordering. As for small to medium messages, message matching is as expensive as the data transfer itself, MPI is unlikely to benefit from the presence of relaxed ordering at the hardware level. Furthermore, on the Cray XE6 Gemini implementation, hardware strict ordering does not scale with the number of cores per node, as indicated in Section III. Thus, the implementation choice to use strict ordering with FMA is likely to lead to severe performance degradation when increasing the number of cores per node.

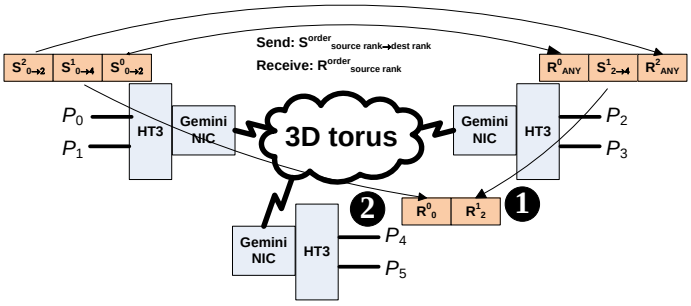

Figure 10. MPI message ordering constraints for deterministic execution.

C. Transfer Size Ambiguity at the Receiver for MPI

The MPI standard allows Send and Recv operations to specify non matching buffer sizes. In this case the sender knows the actual message size, while the receiver provides a buffer *at least* large enough to receive the message. Alternately, message probing allows the receiver to allocate a buffer with the actual transfer size. In either case, this creates a delay at the receiver side in determining what protocol should be used, for instance the choices between FMA or BTE, and whether or not memory registration or bounce buffer allocation is required.

D. Memory Registration and Two-sided Messages

Handling memory registration for two-sided communication buffers is more challenging. The programming model designates the whole memory as private and programmer does not specify any memory region for communication, in contrast with memory declared as shared in PGAS languages. Offloading communication to the hardware can require registration of both the source and the destination buffers. Because of the high cost of this operation, a small amount of memory is preregistered by most MPI implementations and used for eager transfer of small messages. This registered memory is not exposed to the programmer, but is rather transparently used by the runtime. For large messages, memory registration happens on demand, *i.e.*, for application-level supplied memory. Because not all memory can be registered and because failure in memory registration can lead to significant performance loss, many MPI implementations use a registration cache. This cache maintains a subset of recently registered memory segments. A hit in the cache allow earlier initiation of the memory transfer, otherwise memory needs to be registered. The registration decision is more challenging at the receiver side because of message size ambiguity. Researchers have

tried to optimize [11] message matching and provide early registration at the receiver when possible.

For example, Cray MPI exposes two control variables for this registration cache, the first controls the number of entries in the cache and the second controls the maximum size of a registered segment.

E. Performance Analysis of Cray MPICH

The Cray Gemini MPI implementation [12] provides two eager modes and two rendezvous modes for message transfers, switching among them based on message size thresholds, controlled by environment variables. The first eager mode is for very small messages, typically less than 512B, where this threshold depends on the number of ranks. The implementation uses GNI mailbox messages. The second eager mode is for messages that do not fit a mailbox message, but smaller than 8KB. It uses pre-registered MPI buffers and does the transfers using FMA RDMA operations.

For large messages, our measurements tell that Cray MPI delays any memory registration decision until after message matching. The first rendezvous protocol starts with messages greater than 8KB and less than 512KB and it uses receiver end Get-based BTE RDMA. Memory registration, or lookup in the registration cache, is needed for both the sender and the receiver buffers at the beginning of these transfers.

The second rendezvous is used for very large (>=512KB) messages and uses sender Put-based RDMA BTE operations. Memory registration is done in chunks of a maximum size controlled by an environment variable. The implementation strategy obstructs the usage of relaxed ordering or benefiting from having multiple concurrent requests per process. Communication pipelining is also precluded as only registration is overlapped with the transfer.

Furthermore, as discussed in Section III on Cray Gemini Put operations typically achieve only 80% of the bandwidth attained by Get operations. The perceived advantage of this approach for large messages is that it guarantees successful transmission and prevents starvation even with limited memory registration resources.

Figure 11 presents the bandwidth for multiple concurrency levels within the node and also multiple outstanding requests per process (window). We selected four message sizes corresponding roughly to the four protocols used in Cray MPI. The performance trends vary greatly with the message size. In the left side of Figure 11, 8B message performance improves with concurrency up to 8 processes. The performance starts degrading afterwards. The reason for that is due to the use of strictly ordered mailbox messages. We additionally notice that the performance for small messages can be 2× to 4× slower than one-sided communication using relaxed ordering. The performance differences are smaller for 8KB messages,when strictly ordered mailbox messages are used for send/receive matching but relaxed RDMA is used for the data transfer. This leads to mostly monotonically improving performance with concurrency.

Increasing message size to 256KB, we see that performance improves with increased core concurrency. Little improvement can be observed with increasing the number of outstanding requests per rank, which argues its relation with the MPI ordering constraints that obstruct reordering, especially with rendezvous protocols. The performance of MPI is between 33% to 3% below the maximum measured by the interconnect. For UPC, the worst performance is 7% below the peak, but in most cases it is 3% from the peak. Increasing message size further to 2MB, we notice degradation in the performance compared with 256KB because of switching to the Put-based rendezvous protocol. The performance ranges between 33% to 23% below the peak. In this setting, increasing concurrency cannot use the network bandwidth slack because the chunk-based transfer forces registration serialization and little communication pipelining.

In the right hand side of Figure 11 we explore the use of MPI internal buffers for communication and prevent switching to rendezvous protocols[6]. As expected the performance is lower for large messages, but we notice that performance can improve only by increasing message injection concurrency. Having multiple outstanding requests per rank does not help performance.

Notice that the first large message protocol achieves best performance when using 24 cores, while the second rendezvous saturates with one or two cores active.

In the microbenchmark presented in Figure 2 one MPI rank communicates only with one other rank using pipelined ISend/Irecv. Due to the non-overtake rule, message matching cannot be reordered in this case. To allow the implementation more opportunity for reordering we have repeated the experiment having each rank switch communication partners after each message. The performance improves with multiple messages, but this requires multiple ranks per node at the destination. Overall we cannot achieve high percentage of the performance with few ranks per node. Furthermore, given that Cray MPI uses strict ordering for message matching, we have experimented with Open MPI, which uses ID sequence numbers for matching and could possibly use hardware relaxed ordering for tag transmission. Open MPI achieves lower performance than Cray MPI and uses internally FMA mailboxes for matching.

When examining in conjunction the maximum attained bandwidth by one-sided communication in Figure 9 and by two-sided MPI communication in Figures 11, it becomes evident that across all message sizes one-sided can saturate the network using as few as four cores per node, while two sided requires up to 24 cores per node for saturation. This behavior has implication not only in inter-node performance, but also to the intra-node performance. Shared memory programming models, such as OpenMP, achieve better in-node performance by avoiding redundant copying. Using OpenMP reduces the amount of concurrency in accessing the interconnect, thus leading to a tension in using both programming models.

Overall, the Cray MPI implementation achieves lower peak bandwidth than one-sided communication on the Gemini network. A common belief is that most of this performance difference is due to the inability in some cases of the MPI implementation to use RDMA internally. Our results

[6]Setting environment variable MPICH_GNI_LMT_PATH to disabled prevents the use of rendezvous protocols.

show that one-sided communication using non-registered buffers (off-segment) suffers a small performance loss, at most a 6% performance penalty. This one-sided scenario approximates well[7] the runtime behavior of the Cray MPI implementation: either bounce buffers, pre-registered memory due to registration cache hits or pipelined dynamic registration. When examining the performance of MPI in conjunction with in-segment one-sided we observe at most 23% performance penalty in the MPI case. These considerations make us conjecture that the inability to exploit relaxed message reordering in MPI introduces a higher performance penalty than any possibly sub-optimal usage of RDMA and registered memory.

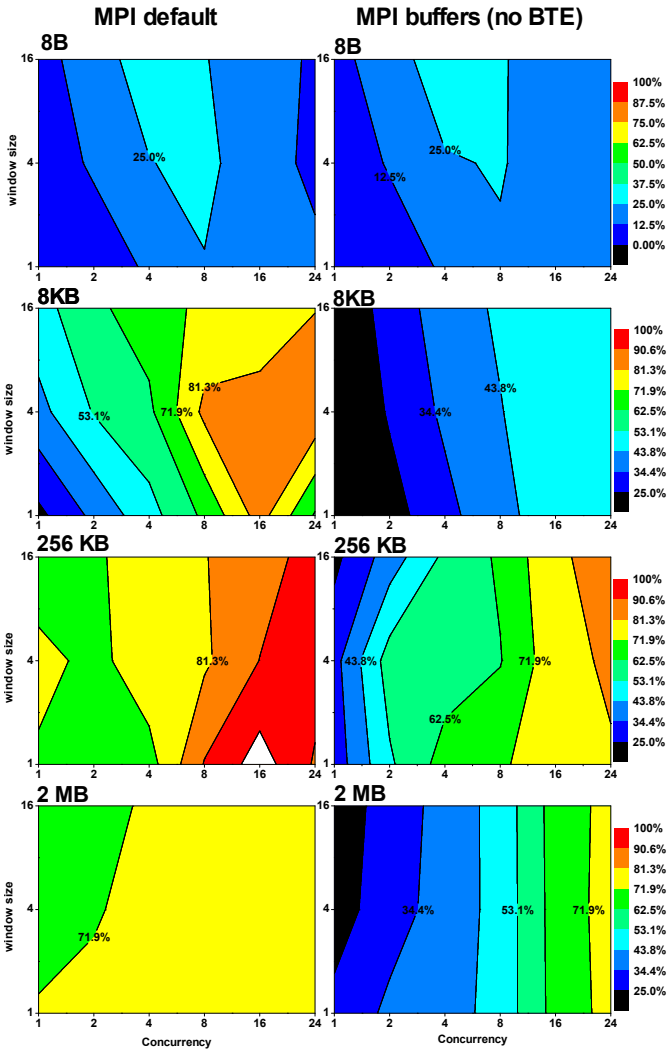

Figure 11. Concurrency *vs.* window size impact on bandwidth performance (MB/s) of two-sided MPI for default and buffer-based executions.

VI. Discussion

This study is a result of our work tuning the performance of the Berkeley UPC/GASNet runtimes on the Cray and IBM architectures. While we provide quantitative information about two particular implementations of UPC and MPI, the question still remains whether the observed trends are caused by a "poor" implementation or by the particularities of the Cray XE6 architecture.

A. MPI Two-sided Implementations and Ordering Relaxation

We surveyed multiple MPI implementations, including IBM mpich on IBM BG/Q using PAMI[8], OpenMPI on Cray systems using GNI, Cray mpich on Gemini/Aries using GNI/DMAPP, *etc*. All these implementations face the choice of providing the semantic orderings through either hardware or software mechanisms. In general, when the interconnect provides both relaxed and strict orderings (through memory consistency or message routing), all MPI implementations use strict RDMA ordering (or deterministic routing[9]) for small messages and during matching send/receive for large messages.

We believe in the optimality of this strategy. The interconnect hardware can sustain high injection rates for small messages using relaxed ordering: we measured 4.3 and 12.5 mega messages/sec for 8 byte transfers on IBM BG/Q and Cray Gemini, respectively. This translates to 368 and 168 CPU cycles for inter-arrival time between incoming messages. The MPI per core message reception rate is determined by the software overhead, which is around 3672 cycles on IBM BG/Q and 1933 cycles on Cray XE6. This already amounts to roughly a $10\times$ slower rate than that delivered by the hardware. At the same time, our results indicate that hardware strict ordering reduces the arrival rate by at most $4\times$ for native communication. Enforcing ordering in software while using hardware relaxed ordering increases the software overhead of processing messages, thus likely to lead to a lower performance.

B. Architectures and Relaxed Ordering

Adaptive routing and relaxed ordering can yield improved performance because it helps in managing contention; they also improve resilience of the system to handle faults. The performance impact reflects in a good measure the architectural system balance.

We experimented with three architectures that support relaxed ordering, Cray Gemini (XE6) and Aries (XC30), and IBM BlueGene/Q interconnects. Of the three architectures, although the oldest, Cray Gemini provides the highest "small message" throughput, about 100 MB/s for 8 byte messages per node. Only Cray Gemini exhibits a significant performance difference when using ordering relaxation even "on few nodes." The other two architectures, IBM BG/Q and Cray Aries, do not exhibit noticeable performance difference except at large scale. BG/Q and Aries interconnects behave similar to Gemini strict ordering, where the performance drops with increased concurrency, but by a smaller margin (at most 10%).

[7]For large messages we can ignore the latency of posting a receive.

[8]PAMI (Parallel Active Messaging Interface) is a portable communication interface introduced by IBM.

[9]IBM MPICH on BG/Q uses deterministic routing for immediate, short, and eager protocols, while allowing relaxed transport of the second phase of rendezvous protocol (after send/recv matching).

These systems represent different points in the architectural design space. They all have a number of cores per node in the low tens but Cray Gemini corresponds to a "resource constrained" network design. The other two systems have better provisioned networks.

The need for relaxed ordering is related to resource allocation in the design. For instance, adaptive routing is not needed for a fully-connected fail-proof crossbar. Likewise, relaxed memory access is not needed on systems that have a high-bandwidth low-latency sequentially-accessed memory. For power-efficiency, hardware designers try to use parallelism and avoid designing for worst-case workload stress, leading to the need for relaxed ordering.

Obviously, systems can be designed to make relaxed ordering not needed, but this is not necessarily the most efficient choice. As the number of cores per node and node performance in High Performance Systems is expected to increase at a higher rate than the network performance, we expect that exploiting relaxed ordering is likely to increase in importance.

VII. Related Work

Optimizing MPI performance using RDMA support has received a large share of attention. Implementation of MPI over InfiniBand is discussed by Sur et al [13] which show how to exploit RDMA for small Send/Recv and protocol messages in addition to large Send/Recv operations. One of their techniques is persistent memory pre-registration for bounce buffers. Woodall et al [14] also describe RDMA optimizations for MPI such as registration caches and pipelining of registration with the data transmission for large messages. Optimizing memory registration is also discussed by Marathe et al [11] which propose an early registration strategy with an associated performance model. Based on our measurements, Cray MPI does not use this approach.

The performance and semantics of MPI-2 one-sided communication has been extensively studied [15], [16], [17] and shown to be restrictive due to implicit inter-process synchronization for Put/Get operations. Potluri et al. [18] show the advantage of one-sided communication for MPI if the implementation is well tuned because of the avoidance of tag matching and sender receiver interactions.

Recently, Dinan et al [17] discuss the performance of the Global Arrays PGAS implemented using MPI-2 one-sided and motivate well the features introduced in the MPI-3 one-sided support. Dinan et al [19] also present and evaluate the MPI-3 one-sided interface on an Intel Xeon cluster connected with the 10 Gb/s Myricom CX4 network. Although not yet well optimized, the semantics of MPI-3 one-sided allow the implementation to take advantage of hardware support for relaxed message order.

Accommodating out-of-order delivery in implementations has been studied for both one-sided and two-sided communication. For MPI, Almasi et al [20] describe handling out-of-order packets on the IBM BG/L network and show better scalability than an implementation with ordered delivery. They evaluate reordering of packets within one message and not message reordering. Handling out of order packets in MPI has also been studied by Balaji et al [21] for the Internet Wide-Area RDMA Protocol (iWARP) over 10-Gigabit Ethernet. Sur et al [13] discuss how to provide message ordering within the MPI two-sided implementation on InfiniBand using sequence numbers. Benefits of message ordering in one-sided MPI-2 are discussed by Vishnu et al [16] for dual rail InfiniBand networks. They evaluate performance on a dual-core cluster and report modest benefits from message reordering. Our results indicate very significant gains are possible for manycores when exploiting hardware support.

Message ordering has started to gather more attention recently, in particular on accelerator based systems. Aji et al [22] examine GPU integrated MPI frameworks and discuss alternatives for buffer synchronization and ordering semantics. In particular, they discuss using MPI communicator or datatype attributes to pass semantic information to the runtime implementation. A similar approach is likely to work when extending MPI implementations with better support for hardware relaxed message order.

The performance on the Cray XE6 system has been evaluated using both micro-benchmarks and whole applications. Vishnu et al [23], [24] present the implementation of the Aggregate Remote Memory Copy Interface (ARMCI) on Cray XE6 using DMAPP with relaxed ordering. Shan et al [25] present a performance evaluation of UPC and MPI benchmarks on Gemini and show applications using single-sided communication outperform those using two-sided paradigm. They evaluate the performance with the default system settings and without evaluating the effect of message ordering.

VIII. Conclusion

Relaxed ordering interconnect exhibits both an opportunity for performance and a complexity for communications library implementers to achieve correctness and to guarantee protocol semantics. In this work, we study the impact of relaxed ordering on Cray XE6 Gemini interconnect. We show that relaxed ordering delivers up to 4.6× performance advantage over strict ordering.

We argue that to fully exploit relaxed ordering the receiver side of a transfer operation should be minimally involved and all target buffers need to be ready for communication early in the transfer (or registered with the interconnect). The semantic of one-sided communication (programming) paradigms can facilitate for the runtime to handle these issues, while two-sided paradigms usually impose restrictions in order to guarantee deterministic execution. The impact of ordering constraints on performance can provide up to 3× advantage for single-sided UPC compared with MPI, for small messages. For Large messages, the difference is up to 30%. More importantly, single-sided UPC can attain a large percentage of the peak performance for most message sizes and concurrency levels. In contrast, two-sided MPI can achieve the same performance only under large message sizes and high concurrency, a configuration that can increase the cost of intra-node communication.

One-sided communication naturally exposes the relaxed ordering power to the application layer leaving the need for strict ordering to special cases. In contrast, two-sided communication tries to exploit the relaxed ordering without

exposing it. This leads to a sophisticated multiprotocol design of the runtime with sub-optimal performance in some instances.

We believe that the findings in this paper can influence how runtime can be optimized to handle relaxed ordering, and how application tuning should be carried out given the understanding of the communication paradigm limits.

ACKNOWLEDGMENTS

All authors from Lawrence Berkeley National Laboratory were supported by the DOE Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] UPC Consortium, http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf.

[2] ——, "UPC Optional Library Specifications- version 1.3," http://upc-specification.googlecode.com/files/upc-lib-optional-spec-1.3-draft-3.pdf, Nov. 2012.

[3] Message Passing Interface Forum, "MPI: A message-passing interface standard version 3.0," www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, Sept 2012.

[4] Hyper transport Consortium, "Hyper transport 3 Specifications," http://www.hypertransport.org.

[5] A. M. Mainwaring and D. E. Culler, "Active message applications programming interface and communication subsystem organization," http://www.eecs.berkeley.edu/Pubs/TechRpts/1996/5768.html, Oct 1996.

[6] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard, "Cray Cascade: a scalable hpc system based on a dragonfly network," *The International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 103:1–103:9, 2012.

[7] OSU benchmarks OMB 3.8, Network-Based Computing Laboratory, Ohio State University, http://mvapich.cse.ohio-state.edu/benchmarks/.

[8] J. Liu, B. Chandrasekaran, W. Yu, J. Wu, D. Buntinas, S. Kini, D. K. Panda, and P. Wyckoff, "Microbenchmark performance comparison of high-speed cluster interconnects," *IEEE Micro*, vol. 24, no. 1, pp. 42–51, 2004.

[9] W.-Y. Chen, D. Bonachea, C. Iancu, and K. Yelick, "Automatic nonblocking communication for partitioned global address space programs," *The 21st annual international conference on Supercomputing*, pp. 158–167, 2007.

[10] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," *The 11th European PVM/MPI Users' Group Meeting*, pp. 97–104, Sep. 2004.

[11] A. Marathe, D. Lowenthal, Z. Gu, M. Small, and X. Yuan, "Profile guided MPI protocol selection for point-to-point communication calls," *The IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pp. 733–739, 2011.

[12] Cray Centre of Excellence, "Optimizing communication for the Cray XE6," *Cray XE6 Performance Workshop-University of Edinburgh*, June 2012.

[13] S. Sur, M. Koop, and D. Panda, "High-performance and scalable MPI over InfiniBand with reduced memory usage: An in-depth performance analysis," *The ACM/IEEE SC 2006 Conference*, pp. 13–13, Nov.

[14] T. S. Woodall, G. M. Shipman, G. Bosilca, and A. B. Maccabe, "High performance RDMA protocols in HPC," *The EuroPVM-MPI 2006*, pp. 76–85, 2006.

[15] W. Jiang, J. Liu, H.-W. Jin, D. K. Panda, W. Gropp, and R. Thakur, "High performance MPI-2 one-sided communication over InfiniBand," *The 2004 IEEE International Symposium on Cluster Computing and the Grid*, pp. 531–538, 2004.

[16] A. Vishnu, G. Santhanaraman, W. Huang, H.-W. Jin, and D. K. Panda, "Supporting MPI-2 one sided communication on multi-rail Infiniband clusters: design challenges and performance benefits," *The 12th international conference on High Performance Computing*, pp. 137–147, 2005.

[17] J. Dinan, P. Balaji, J. R. Hammond, S. Krishnamoorthy, and V. Tipparaju, "Supporting the global arrays PGAS model using MPI one-sided communication," *IPDPS*, pp. 739–750, 2012.

[18] S. Potluri, H. Wang, V. Dhanraj, S. Sur, and D. K. Panda, "Optimizing MPI one-sided communication on multi-core infiniband clusters using shared memory backed windows," *The 18th European MPI Users' Group conference on Recent advances in the message passing interface*, pp. 99–109, 2011.

[19] J. Dinan, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "An implementation and evaluation of the MPI 3.0 one-sided communication," Preprint ANL/MCS-P4014-0113, 2013.

[20] G. Almasi, C. Archer, J. G. Castanos, C. C. Erway, X. Martorell, J. E. Moreira, K. Pinnow, J. Ratterman, N. Smeds, B. Steinmacher-burow, and B. Toonen, "Implementing MPI on the Bluegene/L supercomputer," *Euro-Par Parallel Processing*, pp. 833–845, 2004.

[21] P. Balaji, W. Feng, S. Bhagvat, D. Panda, R. Thakur, and W. Gropp, "Analyzing the impact of supporting out-of-order communication on in-order performance with iWARP," *The 2007 ACM/IEEE Conference on Supercomputing*, pp. 1–12, 2007.

[22] A. Aji, P. Balaji, J. Dinan, W.-C. Feng, , and R. Thakur, "Synchronization and ordering semantics in hybrid MPI+GPU programming," *The 3rd Intl. Workshop on Accelerators and Hybrid Exascale Systems (ASHES).*, 2013.

[23] A. Vishnu, J. Daily, and B. Palmer, "Designing scalable PGAS communication subsystems on Cray Gemini interconnect," *International Conference on High Performance Computing*, 2012.

[24] A. Vishnu, M. ten Bruggencate, and R. Olson, "Evaluating the potential of Cray Gemini interconnect for PGAS communication runtime systems," *High-Performance Interconnects, Symposium on*, vol. 0, pp. 70–77, 2011.

[25] H. Shan, N. J. Wright, J. Shalf, K. Yelick, M. Wagner, and N. Wichmann, "A preliminary evaluation of the hardware acceleration of the Cray Gemini interconnect for PGAS languages and comparison with MPI," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 2, pp. 92–98, Oct. 2012.