Title

Porting GASNet to Portals: Partitioned Global Address Space (PGAS) Language Support for the Cray XT

Authors

Bonachea, D

Hargrove, P

Welcome, M

et al.

Publication Date

2009-05-04

DOI

10.25344/S4RP46

Peer reviewed

Porting GASNet to Portals: Partitioned Global Address Space (PGAS) Language Support for the Cray XT

Dan Bonachea[1,3], Paul H. Hargrove[1,3], Michael Welcome[2] and Katherine Yelick[2,3]
[1]*Computational Research Division, Lawrence Berkeley Nat. Lab. (LBNL)*
[2]*National Energy Research Scientific Computing (NERSC) Center, LBNL*
[3]*Computer Science Division, University of California at Berkeley*

ABSTRACT: *Partitioned Global Address Space (PGAS) Languages are an emerging alternative to MPI for HPC applications development. The GASNet library from Lawrence Berkeley National Lab and the University of California at Berkeley provides the network runtime for multiple implementations of four PGAS Languages: Unified Parallel C (UPC), Co-Array Fortran (CAF), Titanium and Chapel. GASNet provides a low overhead one-sided communication layer has enabled portability and high performance of PGAS languages. This paper describes our experiences porting GASNet to the Portals network API on the Cray XT series.*

KEYWORDS: PGAS, UPC, Cray XT, Portals, GASNet

1. Introduction

Partitioned Global Address Space (PGAS) languages are an emerging alternative to the Message Passing Interface (MPI) for HPC applications development. PGAS languages implicitly involve one-sided communication, in which one process may read or write to the addresses associated with another processor without involving the remote processor. While MPI-2 does include some lesser-used mechanisms for one-sided communication, MPI is primarily a two-sided messaging library. The two-sided model can be awkward for applications involving unstructured communication and it adds complexity to the underlying implementation. Communication between two MPI tasks involves matching the source and destination address information from two different MPI calls. While the matching rules are well defined, they impose a non-trivial burden on the implementation. Additionally the application of the rules to send and receive operations in a given piece of source code is not always immediately obvious, which can frustrate debugging and maintenance.

In a Partitioned Global Address Space model, the access to data across tasks is one-sided, meaning that a communication operation includes both the source and destination address information. This can lead to more efficient implementations and more productive programmers.

Our team at the University of California at Berkeley and Lawrence Berkeley National Lab has developed "GASNet" (Global Address Space NETworking) [1], a portable, high-performance, one-sided communication interface that provides an abstraction of the network and operating system for the implementation of PGAS languages, including our own Berkeley UPC [2] and Titanium [3] language compilers. In addition, the GCC UPC compiler from Intrepid [4], the Co-Array Fortran (CAF) compiler from Rice [5] and Cray's Chapel [6] compiler are using GASNet for their network communications. Recently Cray released their own compilers for UPC [7] and CAF [8] on the XT, and they too are using GASNet.

When the U.S. Department of Energy began to purchase Cray XT hardware, the Berkeley UPC compiler was ported to run applications under Catamount, but only by using GASNet's "mpi-conduit" for communications. While GASNet can run over MPI for maximum portability, this significantly reduces performance relative to an implementation for the native network API. This paper describes our experience developing "portals-conduit", GASNet's native implementation over the Portals [9] API for the Cray XT series. We present some background on the relevant hardware and software followed by a chronological account of the port; a performance comparison of mpi- and portals-conduits and some closing remarks.

2. Background

The Cray XT product series [10] form a family of scalable compute platforms composed of Opteron processors on a custom 3D torus interconnect. A custom network interface chip called "SeaStar" implements the 3D router, a HyperTransport interface to the host main memory, and DMA engines that transfer data to and from host memory and the network. The SeaStar also contains a PPC440 processor and local scratch memory to help off-load the communication processing from the host.

GASNet provides language implementers with one-sided communications operations and a remote procedure call mechanism. GASNet is intended as a target for compilers and a limited group of expert programmers, as opposed to end-users. Therefore, this paper will not attempt to teach the details of the GASNet interface. Instead we begin with a broad overview of GASNet and then introduce details of GASNet and Portals as needed to describe the porting issues.

We designed the *implementation* of GASNet with a layered approach to ease porting to new platforms. The lowest levels include build infrastructure and network-independent support specific to the O/S and CPU. Above the infrastructure layers our term for the support for a given network is a "conduit". The implementation of GASNet's public API is split into two levels: the "Core API" and "Extended API". The Core consists of the subset of GASNet calls that must be ported to any given network, and includes program startup and a simple remote procedure call mechanism, known as Active Messages (AM). The Extended API includes a wide range of data-movement calls such as "Put" and "Get" operations in a variety of blocking and non-blocking variants. GASNet provides a reference implementation of the Extended API in terms of calls to the Core API. This design allows an implementer to get started with just an implementation of the Core API, and then incrementally replace portions of the Extended API's reference implementation with ones customized to the network.

Our goal was to implement GASNet over Portals, the lowest-level public interface to the Cray XT's SeaStar communications hardware. We start by moving the GASNet mpi-conduit to the XT under Catamount, which involved modifications of the lowest level infrastructure layers. The next step started with an analysis of Portals. The Portals API was designed with MPI implementations in mind, and therefore includes features well aligned with the send/recv matching required by MPI. Portals supports Remote Direct Memory Access (RDMA) for efficiency in the movement of data, and all data transfers in Portals are accomplished via Put/Get style operations that utilize a sophisticated mechanism for determining the remote address.

3. Portals-conduit Stage 1: MPI Hybrid

While the layers of implementation in GASNet were designed to allow a new platform to begin with a port of just the Core, we chose to pursue a different approach with the initial implementation of portals-conduit. Since mpi-conduit uses the reference implementation of the Extended API, portals-conduit began by replacing this with portals-based implementation of the Extended, while preserving the MPI-based Core implementation from mpi-conduit. This allowed us to begin work immediately on the portions of the code that would make the greatest impact on performance of PGAS applications.

3.1 Addressing in GASNet vs. Portals

The most important one-sided communication operations in the GASNet Extended API are "Put" and "Get" operations to move data between two processes (a process is a "node" in GASNet terminology, analogous to an MPI "task"). There are various blocking and non-blocking variants, in addition to collective and vector/indexed/strided operations. However, all the Put and Get operations have in common that the caller specifies a local address and length, plus a remote node number and a remote address[1]. The Put and Get operations are all semantically one-sided, in that there is no corresponding GASNet call on the peer node to receive the data of a Put or to provide the data for a Get.

While Portals provides an efficient RDMA mechanism for movement of data, it is not based directly on virtual addresses as in GASNet. In Portals, one process may allow another to access portions of its address space by creating a Memory Descriptor (MD) that gives the address and length of the region in memory, along with other attributes. The arguments to the Put or Get initiation operation in the Portals API include a local MD and an offset relative to its base. For the remote address Portals allows the initiating node to specify the target node and an offset relative to the base of an *undetermined* MD, and some matching arguments including a "portal index" and a 64-bit word of "match bits".

At the Portals target node, three layers of data structure determine the target MD of any incoming Put or Get. The first is a table indexed by the portal index, which separates distinct clients of Portals[2], and selects an ordered list of Match Entries (ME) on the target node. Each ME includes two 64-bit words ("match bits" and "ignore bits") and an MD handle. The Portals implementation traverses the

[1] GASNet supports a choice of "Segment" configurations in which the remote address is either restricted to a specific portion of the remote address space, or is unrestricted. In all configurations the local address is unrestricted. At this time portals-conduit only supports the restricted case for remote addresses.

[2] GASNet uses just two portal ids, and they are distinct from those used by the MPI and SHMEM implementations, allowing mixed-mode applications.

MD	PTE	Match Bits	Ops Allowed	Offset Mgt.	Event Queue	Description
RAR	RAR	0x0	PUT/GET	REMOTE	NONE	Remote segment: dst of Put, src of Get
RARAM	RAR	0x1	PUT	REMOTE	AM_EQ	Remote segment: dst of RequestLong payload
RARSRC	RAR	0x2	PUT	REMOTE	SAFE_EQ	Remote segment: dst of ReplyLong payload Local segment: src of Put/Long payload, dst of Get
ReqRB	AM	0x3	PUT	LOCAL	AM_EQ	Dest of AM Request Header (double-buffered)
CB	AM	0x3	PUT	LOCAL	SAFE_EQ	Catches any AM Requests that miss all ReqRBs, for detection of fatal credit management errors
ReqSB	AM	0x4	PUT	REMOTE	SAFE_EQ	Bounce buffers for out-of-segment Put/Long/Get, AM Request Header src, AM Reply Header dst
RplSB	none	none	N/A	N/A	SAFE_EQ	Src of AM Reply Header
TMPMD	none	none	N/A	N/A	SAFE_EQ	Large out-of-segment local addressing: Src of Put/AM Long payload, dest of Get
SYS	AM	0x5	PUT	LOCAL	SYS_EQ	Startup, shutdown, and credit redistribution

Table 1. Memory Descriptors in portals-conduit

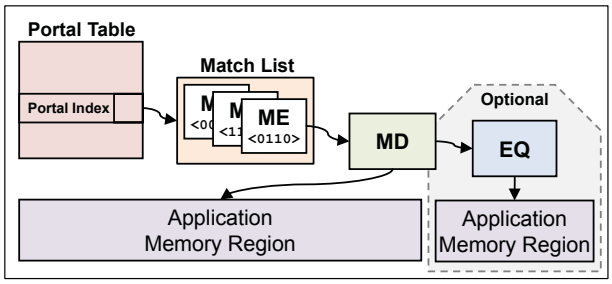

Figure 1. Portals Message Processing

ordered list of MEs until the match bits provided by initiator and ME are the same after discarding the ME-provided ignore bits. If a matching ME is found and the associated MD meets some additional criteria, then the initiator-provided offset is added to the Base of the MD to yield the target-side address of the Put or Get. Portals message processing is depicted graphically in Figure 1. Table 1 lists all the MDs utilized in portals-conduit.

While match bits provide for dynamic use of MEs to support posted receives in MPI, implementing GASNet Puts and Gets needs only the ability to address a single contiguous "segment" on the remote node. GASNet establishes this access at startup using one portal id, "RAR_PTE", with multiple MEs to distinguish different operations (as will become clear later). These MEs have a common set of ignore bits that exclude all but the least significant 4-bits from matching. For addressing of the GASNet segment at the target, the "RAR" (Remote Access Region) is an MD covering the range of addresses in the GASNet segment, associated with an ME that matches "RAR_BITS". This MD is configured to allow Put and Get access, to remain persistently available, and not to generate any events (described later) on the target node. Using a table established at startup containing the base address of each node's GASNet segment (and thus RAR) the initiator of a Put or Get can name the appropriate destination using RAR_PTE for the portal index and RAR_BITS for the match bits, while the remote offset is the remote address minus the base address.

On the initiator there are two distinct possibilities for the local address: it either lies inside the GASNet segment or outside. We will return to the out-of-segment case later. For the case of addresses inside the segment a second MD, known as "RARSRC" (RAR SouRCe), is used. This MD covers the same range of addresses as "RAR" and is passed as the local MD for Portals Put and Get operations with a local offset computed as for the remote case. However, RARSRC is different from RAR in that it is configured to generate Portals events used to implement GASNet's completion semantics, as described below.

3.2 Completion of GASNet operations

GASNet provides a variety of blocking and non-blocking operations to support the needs of PGAS clients. The blocking operations in GASNet return after data movement is complete (remote completion), while the non-blocking ones have associated "sync" mechanisms to poll or wait for remote completion. The non-blocking Put functions are further divided into "non-bulk" ones where the initiation call may return as soon as the local data is safe to overwrite (local completion) and "bulk" ones that may return immediately (while the source buffer is still in use). In order to implement the various options over Portals, one needs to understand the Portals event mechanism.

In Portals each MD may have an associated Event Queue (EQ). A given operation can trigger creation of several possible events on the EQs associated with the MDs at the initiator and target. A combination of per-operation flags and MD flags control which events are generated. A Portals call to initiate a Put or Get will return immediately without regard to the state of the transfer. To know when a given Portals operation has completed, either locally or remotely, one must process the Portals EQs.

Portals Event	GASNet Event
SEND_END	Local completion of Put
ACK	Remote completion of Put
REPLY_END	Completion of Get

Table 2. Portals Events for a Put or Get

From the perspective of completion, the most complex operation in GASNet is the non-blocking, non-bulk Put operation. Once it initiates the Portals-level Put operation, it must wait for local-completion before it may return, and a later sync operation must test or wait for remote-completion. Bulk operations are simpler because they may return without concern for local completion. Gets are even simpler because there is no distinction between local and remote completion. In our implementation for other low-level communications APIs, GASNet is sometimes forced to wait for remote completion when there is no independent indicator of local completion; or GASNet may need to implement an acknowledgment when there is no indication of remote completion. With Portals, however, there is a one-to-one mapping from Portals events at the initiator to GASNet's completion semantics, as summarized in Table 2.

Since there can be multiple GASNet-level operations in flight, there is some additional work that GASNet must perform to associate a Portals-level event with the GASNet-level operation. As was mentioned previously, the Match Entry for the RAR MD ignores all but the least-significant 4-bits. Since all match bits of the original operation are preserved in any corresponding EQ entry, this is an ideal place to store the information needed for establishing this correspondence. Four additional bits are used to store the operation type (including bulk Put, non-bulk Put, and Get[3]). One byte is used for a thread id[4] and two bytes for a GASNet "eop addr". The thread id and eop addr are existing GASNet abstractions in the reference-extended implementation and provide for a compact naming of a specific outstanding GASNet-level operation. These fields account for a total of only 32 of the 64 Portals-provided match bits; the remaining 32 became useful in implementing additional features, as will be described in subsequent sections.

3.3 Out-of-segment access

In the case of a local address outside of the GASNet segment, Portals still requires GASNet to name a local MD when initiating communication, but RARSRC covers only the GASNet segment. GASNet handles this in one of two ways depending on the length of the operation: bounce-buffers or TMPMD. For small payloads, GASNet maintains a pool of buffers covered by a single "ReqSB" MD configured very much like RARSRC. A small GASNet-level Put with out-of-segment source address is copied into a bounce-buffer and the Portals-level Put is issued with the bounce-buffer as the source. Similarly a small GASNet Get with an out-of-segment destination address is issued to Portals as a Get with a bounce-buffer as the destination and the data is copied to the caller's buffer after the Portals-level Get is complete (the REPLY_END event).

For out-of-segment transfers larger than a tunable threshold, GASNet switches to dynamic creation of MDs. GASNet creates a "TMPMD" Memory Descriptor to cover the requested range of local memory, and destroys the MD when the Portals-level Put or Get is completed. There is some (OS-dependent) software overhead associated with MD creation and destruction, hence the tradeoff between this penalty and the cost of copying the payload through a bounce buffer on the initiator. The implementation can manage many TMPMDs to support multiple outstanding non-blocking operations.

The TMPMD and ReqSB MDs are configured to place associated events on the same EQ as the RARSRC used for in-segment local address. Therefore, handling of GASNet completion semantics is altered only slightly relative to the in-segment case. In particular if using a bounce-buffer for a Put then the copy to the bounce buffer is sufficient for local completion, without waiting for a Portals SEND_END event. There is no need to use any match bits to identify in-segment vs. bounce-buffer or TMPMD, as this is easily distinguished using the MD handle in the EQ entry.

Regardless of the choice between bounce-buffer or TMPMD, the associated resource is returned to its free pool upon completion. A specifc TMPMD can be identified trivially because the MD handle is stored in the EQ entry. Similarly a bounce-buffer address can be reconstructed from the base address of its MD and the local_offset in the EQ entry.

3.4 Finite Resources

The description so far has ignored finite resource limitations. However, Portals EQs are created with a caller-specified finite capacity that must not be exceeded. For the Put/Get code this is easily dealt with using a semaphore-type construct that ensures that GASNet will not initiate a Portals-level operation until it has acquired sufficient "send tokens" to account for the events that the Portals operation will generate. If the counter of available send tokens is zero, then portals-conduit will poll the EQ to retire operations. A send token is freed when processing the event marking the completion of the associated Portals operation.

4. Portals-conduit Stage 2: Native AM

The implementation described in the previous section used a portals-specific implementation of the GASNet Extended API to achieve significant improvements in applications level performance[5]. However, when the native

[3] Additional operation types are used to support additional features.

[4] While threads were not supported under Catamount, they are in CNL. GASNet supports threads, which permits a hybrid execution model involving threads within a node and GASNet between them.

[5] A performance comparison is presented in Section 6.

Extended API implementation was completed, portals-conduit still relied on the MPI implementation of the GASNet Core, including job startup/teardown and Active Messages. The next major step for portals-conduit was to implement these pieces directly.

The GASNet Core API includes process startup and exit, which were previously implemented in terms of MPI_Init and MPI_Finalize. The implementation of process startup under Portals on Catamount was not too difficult to derive from documents, but the implementation of MPI_Init in Open MPI's port to Red Storm was the easiest place to find the necessary logic. While some non-trivial work was required to make process exit as robust as possible, it was based on designs used in previous GASNet conduits and there is nothing of interest to describe here.

4.1 AM Overview

The Active Message interface in GASNet provides a simple remote procedure call mechanism. At startup a GASNet client establishes a table of pointers to functions, called "Handlers". An AM consists of a required index into the handler table at the destination, optional arguments and optional payload. A node may send an AM "Request" to any node and the "Request Handler" is determined by the handler index and executed with the arguments and payload (if any). An AM Handler is prohibited from blocking indefinitely and is allowed to make calls to only a subset of the GASNet API. The only GASNet communication calls permitted within Handler context is a Request Handler may generate a single optional AM Reply to the node initiating the Request; a Reply Handler may not perform *any* communication. These restrictions are derived from the original Berkeley AM [11] specification and combine to ensure the implementation may be efficient in terms of both code complexity and minimal resource usage. The "One-Request + Zero-or-one-Reply" requirement helps ensure deadlock-free management of resources.

4.2 Request Receive Buffers

Some GASNet conduits dedicate a number of fixed-size buffers to receive incoming AMs, and in some cases the buffers are associated with specific peers leading to non-scalable memory consumption. However, under Portals we use a "Locally Managed" MD for receiving AM Requests. An MD as described for Put and Get operations is "Remotely Managed" and forms the destination address by adding an initiator-provided offset to the base address of the MD. In a Locally Managed MD, the initiator's offset is ignored and arriving data is automatically concatenated by Portals starting at the Base of the MD. This Portals feature allows reception of a sequence of variable-sized messages into a single long buffer with a minimum of waste[6]. GASNet uses multiple instances of this "ReqRB" (Request Recv Buffers) MD linked into a portals table entry "AM_PTE" distinct from RAR_PTE. Like other MDs linked to the portal table, they too have an ME that ignores all but the four least-significant match bits. The multiple instances of ReqRB implement a double-buffering (or triple) scheme, allowing an empty one to automatically begin accepting messages when the arrival of a message finds too little space in its predecessor. When GASNet finds a full ReqRB MD, it is unlinked from the ME list, and is returned to the end of the list when GASNet has processed all of the Requests it contains.

4.3 Sending an AM Request

AM Requests are constructed in memory by the initiating node, allocating memory from the same "ReqSB" (Request Send Buffers) MD that is used for bounce-buffers in the Put/Get code. Since GASNet AMs may have at most a single reply, and the maximum size of a reply is known at compile time, the buffer used to construct the Request is also used to receive the corresponding Reply. Because of this, if an AM Request Handler completes without sending the optional Reply, our implementation generates one implicitly to ensure the requester's buffer can be recycled.

GASNet defines three categories of AM: Short, Medium and Long. They have in common that the caller provides a handler index and up to 16 optional arguments of 32-bits each. The category, handler index, arguments and implementation-specific metadata are collectively known as the AM "header". The categories differ in the treatment of the "payload".

We can begin to understand the AM-over-Portals implementation by examining sending and receiving a Medium Request. In the Medium case, the caller specifies a payload by the local address and length, whose contents are to be delivered into a conduit-managed buffer at the target for use by the AM Handler. The implementation needs only the lowest 4 of the 64 match bits for target MD selection and uses the upper 32 to store the local offset of the buffer (relative to the ReqSB) which will be used for delivery of the Reply. The remaining 28 bits are used to pack the category, handler index, argument count and payload length, plus flow control credits to be described later. Additionally, the Portals-level Put operation used to send the AM can carry 64-bits of "header_data" for delivery to the EQ entry, which is used for other header fields or up to two 32-bit handler arguments. Any arguments beyond those are written to the buffer allocated from the local ReqSB, followed by a copy of the caller's payload[7]. This buffer is then Put to the recipient with the portal index and match bits required to target the ReqRB. Flags are passed to Portals that suppress generation of the ACK event and the SEND_END event is ignored.

[6] GASNet requires 8-byte alignment of Medium payloads. Padding inserted by the sender to ensure this alignment is the only per-message wasted space in the ReqRB MD.

[7] Padding may be added before and after the payload to maintain 8-byte alignment of both the payload and the next header.

A Short request differs from the Medium only in having no payload. Long requests are more complex. In a Long, the caller provides not only the local address and length of the payload, but also a remote destination address in the GASNet segment at the target for delivery of the data (semantically, an AM Long may be thought of as a Put which invokes an AM Short at the target upon arrival of the data). For sufficiently small payload, a Long is sent the same way as a Medium, with the addition of the target address to the header, and the receiver will copy the payload to the given address. For payloads too large for this "packed long" approach, two Portals-level operations are required to move the header and payload to their disjoint destinations. The header is sent as before with addition of a generated unique integral "Lid" (Long ID), the use of which is described below. However, in order to address the GASNet segment and generate an event at the target, a different MD is required for the payload. This MD is called "RARAM" and is identical to RARSRC except for having a separate EQ for reasons described below. The AM Long payload is sent with a Portals Put to RARAM that passes the Lid in the Portals header_data. Other than passing this Lid for use in the remote event, the Long payload is transferred using the same code as the GASNet-level Put. This includes stalling for local completion on a RequestLong like a non-bulk Put[8].

4.4 Receiving an AM Request

The ReqRB MD is configured to generate a PUT_END event when an AM Request header arrives. So when GASNet sees a PUT_END event on this MD, it will begin processing the AM header. In the case of the Short and Medium categories of AM the Request handler may run as soon as sufficient resources can be allocated to generate the Reply. The same is true for a packed Long, but in the case of a Long that was not packed there is also a PUT_END event for the payload. Since Portals does not guarantee the order these will be processed (especially in the case of a threaded GASNet client), we use the Lid included in both events to match the two events and ensure the handler is run only when both the header and payload have arrived.

4.5 Sending and receiving of AM Replies

In order to avoid deadlock, the sending of a Reply must not require the implementation to block for unbounded time waiting for resources; therefore it does not process additional Requests (since their continued arrival could starve the pending Reply indefinitely). As was mentioned above, portals-conduit collects the resources needed for a worst case Reply before running a Request handler. If necessary this will stall waiting for Portals events that free up resources used by in-flight operations. Doing this prior to executing the Request ensures the Reply path will never block waiting for resources.

To avoid reentering the request-handling path, portals-conduit utilizes two separate EQs for the events generated at the target by arriving AMs. The events generated by arrival of a Request in RARAM are on a distinct "AM_EQ" used for no other purpose, while the Reply-generated events are on the same "SAFE_EQ" as all others we process. In this way we ensure that when a Long Reply stalls for local completion, it will do so while polling only the SAFE_EQ which is guaranteed not to trigger synchronous execution of additional AM Request handlers.

Separation of EQs for Request and Reply arrivals requires that their Portals-level operations address distinct MDs. Therefore sending of a Reply is very similar to a Request, but differs in how the data is addressed. The Reply header and any Medium (or packed Long) payload is not sent to the ReqRB MD, but instead back to the same portion of the ReqSB MD that was used to construct the Request (using the offset held in the upper 32-bits of the Request header's match bits). Similarly, the payload Put of a Long is addressed not to the RARAM MD but to the RARSRC MD. So, the arrival of a Reply at its destination generates a PUT_END event associated with the ReqSB MD, while a Long payload will generate one associated with the RARSRC MD. Additionally, the reception of a Reply differs in that there is no need to allocate resources before running the handler.

4.6 Flow Control

Like the Put/Get code, the AM Request code allocates send tokens to account for the locally-generated EQ entries. However, AMs also generate PUT_END events on *remote* nodes upon arrival of AM headers and Long payloads. Without some flow control to limit the arrival of these events, one may exhaust the space in either the AM_EQ or the ReqRB. The available space depends on the message backlog and servicing rate at the target, and thus cannot be managed on the sender alone. For this one needs a mechanism to ensure that the total number of arriving AMs from all peers will not exceed the space in the AM_EQ or ReqRB.

Initially, portals-conduit adopted a standard credit-based flow control scheme for managing the remote resource consumption. In this scheme, the AM Request code must acquire one or more "credits" to account for the space it will consume on the target node (in addition to the send tokens used to account for space in the local-side SAFE_EQ). Each credit represents 256 bytes of ReqRB space plus one AM_EQ event. An AM Short requires one credit, whereas Medium and Long messages may require multiple credits depending on the payload size. Credits are implicitly returned by Replies and freed when processing the corresponding PUT_END event on the ReqSB. This is

[8] GASNet also provides a RequestLongAsync which, like the bulk Put, is not required to wait for local completion.

the motivation for sending an implicit Reply if a Request handler does not send one explicitly.

4.7 Advanced Flow Control

The description of flow control above assumes a known value of how many credits a given node has for sending AM Requests to a given peer, but does not address how that value is chosen. For each credit granted there must be corresponding space in the AM_EQ and ReqRB at the target, arguing for smaller values to avoid excessive memory consumption at larger scales. On the other hand, a small number of credits will limit the number of AM Requests a given node may have un-acknowledged by a given peer, potentially reducing performance of AM-intensive message streams[9], which argues for larger values. Initially our implementation chose a simple static partitioning of the credit resources in which a suitably large AM_EQ and ReqRB were created and each peer was granted an equal share of the available credits. The static partitioning was easy to implement and with suitably large parameters could grant every peer sufficient credits to keep many AM Requests outstanding to cover their latency in the worst case. However, this came at the cost of granting equal credits (and thus allocating equal space) for every peer regardless of the actual usage imposed by the communication pattern.

We have observed that few applications are written to require AM-intensive communication between every node pair, and that many nodes receive significant bursts of AM Requests from very few peers (such as nearest-neighbors). Given the amount of memory that was required to give equal credits to all peers for a large scale run, we turned our attention to a dynamic credit allocation scheme that would take advantage of the common case of few AM-peers to significantly reduce the memory required for the AM receive processing while still allowing a large number of outstanding AM Requests between peers that could benefit.

The dynamic credit management algorithm works by granting each peer at startup a constant number of credits that sums to *less* than the allocated resources (to ensure that every node may send *some* AM Requests), but holds some resources in reserve to grant to those peers who may need more. The key idea is that a node that stalls for lack of credits when trying to send a Request can inform the target peer of how many additional credits would have allowed it to send the Request without stalling (using just a few bits in the header of the Request). The target of the AM Request may then consult its current resource availability and include in the header of its Reply a "loan" of additional credits to the needy peer. In this way only those peers that would benefit from a greater-than-default number of credits will consume them. While a complete description of the algorithm is beyond the scope of this paper, it includes mechanisms to allow redistribution of credits if the application's communications patterns change over time.

[9] AM-intensive operations in GASNet include barriers and some other collectives, while the Berkeley UPC compiler uses AMs for remote lock/unlock and certain memory allocation operations.

5. Portals Conduit Stage 3: Firehose

Among the most recent developments for portals-conduit is the use of our "Firehose" library [12] to replace the TMPMD mechanism. The TMPMD mechanism adds the synchronous overhead of creating and destroying an MD to the latency of every out-of-segment Put or Get operation that exceeds the bounce-buffer threshold. When TMPMD was first developed CNL was not yet available outside of Cray, and MD creation and destruction were negligible on Catamount. However, when we began running on CNL the costs were no longer trivial[10].

The Firehose library was originally developed as part of GASNet's gm-conduit support for Myrinet hardware. It was later generalized to be conduit-independent and used in GASNet's InfiniBand and LAPI-RDMA support. The purpose of Firehose is to take advantage of temporal and spatial locality of reference to amortize the cost of dynamic memory registration. As with TMPMD, a Put or Get with an out-of-segment local address obtains an MD for use in the corresponding Portals-level operation. However, rather than unconditionally creating a new MD, Firehose caches recently used MDs and in the common case will reuse an existing one. Similarly when the Portals-level operation is complete, a reference count is decremented rather than destroying the MD. The Firehose library uses a simple LRU scheme for eventually destroying MDs when necessary to limit the total amount of memory registered.

In addition to LRU replacement to limit MD usage, Firehose will notice if the MD that would be required for a given operation does not yet exist but is adjacent to, or overlaps, one or more existing MDs. In such a case, we perform coalescing: creating a new MD that is the union of the existing MDs and the present operation[11], up to some maximum size. We also mark the now-redundant smaller MDs for destruction on the next call. Through this continuous adaptation, the Firehose library will quickly discover the out-of-segment working set of most applications and encompass it with a small set of MDs after a relatively short number of calls.

The conversion of portals-conduit from TMPMD to Firehose was not without complications. The fact that Firehose-created MDs have a maximum size meant that

[10] The SeaStar hardware requires memory it addresses be "pinned" by the OS, which is ensured at MD creation. All memory in Catamount is pinned, but CNL must traverse its memory management data structures to ensure pinning.

[11] While TMPMDs were created for the exact range of addresses to be transferred, firehose expands all requests to page-sized granularity to take advantage of spatial locality of reference and to assist this coalescing.

GASNet-level Put and Get operations that had always required exactly one Portals-level operation under TMPMD can require multiple ones under Firehose. For the Put and Get operations this was not a problem and only minor bookkeeping changes were required. For instance, the upper 32-bits of match bits are used to determine which Firehose is to be released when an operation is complete. However, for the AM Long payload a naïve transition from TMPMDs to Firehose would have meant that an unpredictable number of PUT_END events would be generated, requiring an unpredictable number of credits. This was resolved by reducing the maximum size of a Long payload under portals-conduit to what would fit in a single Firehose-managed MD (allowing for worst-case misalignment to page boundaries)[12], eliminating any variability in credits required.

6. Performance Results

In this section we present some microbenchmark results to illustrate the benefit that came from implementing GASNet natively over Portals rather than relying on mpi-conduit to run on the Cray XT series.

6.1 Experimental Platform and Methodology

All performance numbers were obtained in a single two-node batch job on Franklin [13], the quad-core Cray XT4 at NERSC. Only a single core was used on each node. Environment modules loaded included PrgEnv-gnu/2.1.50HD and xt-mpt/3.1.2.

All data state the mean per-operation performance measured by timing 10,000 consecutive iterations of the subject operation. In the case of non-blocking operations this time includes initiating all 10,000 operations followed by blocking for completion of all 10,000. All sizes and bandwidths are reported in units $K=2^{10}$ and $M=2^{20}$.

All MPI results use the Cray MPI-1 message passing library, which is implemented over Portals. We do not consider MPI-2 RMA because as discussed in [14], that API is semantically unsuitable for use as a PGAS compilation target. The same paper describes the design of mpi-conduit and the semantically-imposed costs of implementing one-sided put/get over MPI message passing.

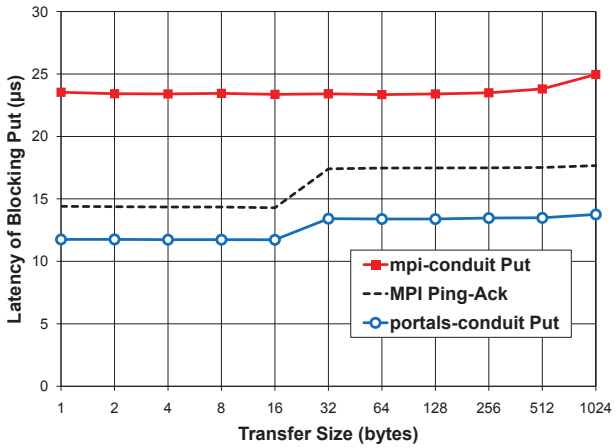

Figure 2. Small Transfer Latency

6.2 Small Transfer Latency

In Figure 2 we examine the latency of a blocking Put operation (lower latency is better). The upper line shows the performance of mpi-conduit. The lower line shows the latency of portals-conduit. For sizes up to 16 bytes, the improvement is roughly a factor of two. The discontinuities at 16-bytes in the lower two lines are because the SeaStar hardware on the Cray XT can place the first 16 bytes of Portals message payload in the same wire-level packet as its own header, resulting in nearly constant cost up to 16 bytes.

The significant difference between the MPI and Portals implementations of the GASNet-level Put comes from two sources. The first is that the GASNet completion semantics require an acknowledgement that the data has reached remote memory. For portals-conduit this is available directly from the Portals ACK event, while mpi-conduit must perform an additional MPI-level communication for the acknowledgement.

The remaining difference between the implementations is that in mpi-conduit both the initial data movement and the acknowledgement are implemented over GASNet's Active Messages, and pay some costs associated with the semantic mismatch of implementing one-sided operations over two-sided message passing[13]. To facilitate comparison, we include an additional benchmark: the "MPI Ping-Ack" (dotted line). This is a simple benchmark written in MPI that sends an n-byte message and a zero-byte reply. This is the minimal MPI-level communication required to simulate the data movement and synchronization of a GASNet-style Put operation and is a lower bound on the latency achievable by mpi-conduit. Note that portals-conduit out-performs this MPI benchmark by a significant margin.

[12] This limit can be controlled by an environment variable, but the default value allows for Long payloads up to 124K.

[13] For instance, the lack of a discontinuity in the latency results arises because the combined PutGet-over-AM-over-MPI headers exceed 16 bytes. See reference [14] for further details.

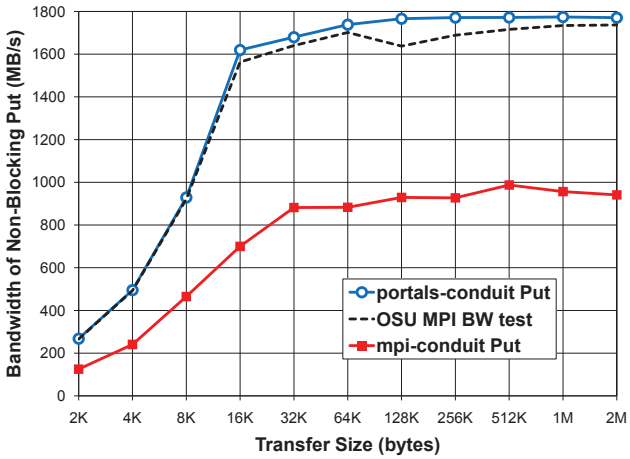

Figure 3. Large Transfer Bandwidth

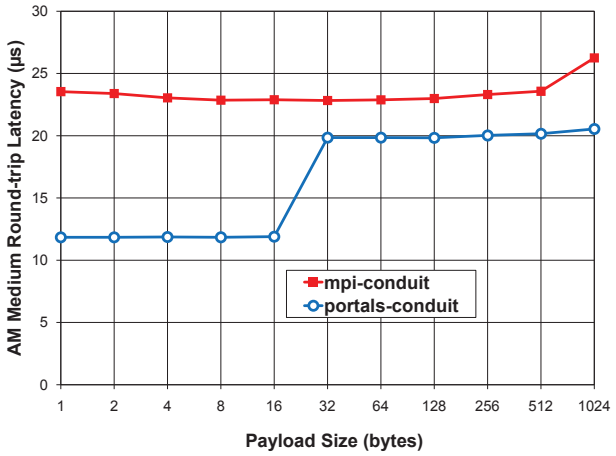

Figure 4. AM Medium Latency

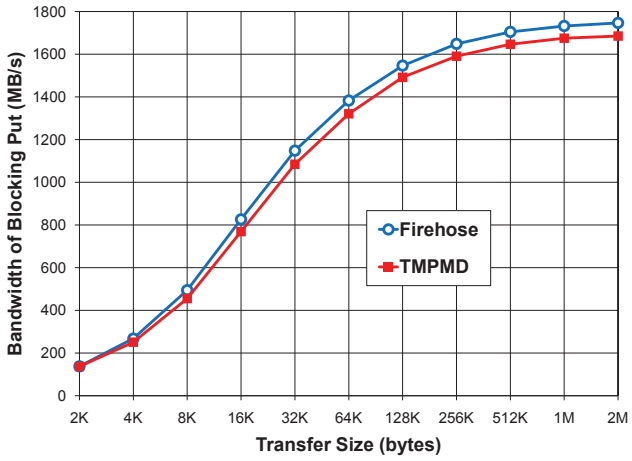

Figure 5. Firehose vs. TMPMD for Blocking Put

6.3 Large Transfer Bandwidth

In Figure 3 we examine the bandwidth of a non-blocking Put operation (higher bandwidth is better). This benchmark initiates a large number of non-blocking Puts before blocking for any completion, expecting that the costs of communication initiation can be overlapped with communication. As in the small-transfer latency case, we find portals-conduit yields roughly a factor two improvement over mpi-conduit across the entire range measured.

At larger transfer sizes, the primary bandwidth difference arises from the fact that mpi-conduit requires two in-memory copies of the payload (one at each end) to implement one-sided Put over two-sided MPI message passing, whereas Puts in portals-conduit can directly leverage the zero-copy Put operations offered by Portals. Again we provide results (dotted line) from a comparable MPI benchmark: "osu_bw"[14]. As seen before, GASNet's portals-conduit is able to meet or exceed the performance of the MPI benchmark.

6.4 Active Message

In Figure 4 we see the motivation for replacing the MPI-based implementation of Active Messages with a portals-based one. This figure shows the time required for a round-trip of AM Medium (a Request and a Reply), each with zero arguments and a payload of the indicated size. For this benchmark lower results are better. The results are very similar to those for the blocking Put latency (Figure 2) and the two figures use the same scales to ease comparison. As with the Put latency, there is roughly a 2-fold difference up to 16 bytes of payload. Unlike the Put latency, the communication is bidirectional, leading to a more significant discontinuity at 16 bytes. While this leads to a less significant advantage for portals-conduit between 32 and 512 bytes than was seen in the Put case, the cost mpi-conduit pays for additional in-memory copies becomes evident between 512 and 1024 bytes.

6.5 Firehose vs. TMPMD

Finally, Figure 5 demonstrates the performance gain of replacing the TMPMD mechanism with the Firehose library. This figure shows large-transfer bandwidth (higher is better), as did Figure 3, and they are plotted on the same scales. In this case, however, the operation timed is a sequence of large *blocking* Puts with an out-of-segment source address. The *same* source address is used for every Put operation, which is the ideal case for Firehose. Because a sequence of blocking operations cannot benefit from overlap of injection costs with data communication, the reduction in injection cost is evident in this microbenchmark. In absolute terms the difference is growing with transfer size because the time to create an MD grows with size; the TMPMD implementation pays this cost for *every* operation while the Firehose implementation

[14] This benchmark is distributed by Ohio State University with their MVAPICH implementation of MPI.

amortizes this cost over all the operations at a given transfer size.

In this figure the lower line represent the use of TMPMD, and the upper line is Firehose. The results show that using Firehose consistently achieves an equal or higher bandwidth than with TMPMD. Asymptotically the improvement is approximately 3.5%, and the greatest improvement of 8.5% is seen at 8KB.

7. Conclusions and Future Work

As can be seen from the performance results in the previous section, the native port of GASNet to Portals was clearly worthwhile from a performance perspective. The microbenchmark results show this port yields approximately half the latency for small transfers and twice the bandwidth for large ones. Additionally, comparisons to MPI microbenchmarks with the same communications patterns (but without the full GASNet semantics) show that no amount of tuning of mpi-conduit could have been expected to produce the performance of portals-conduit. Overall we feel it was well worth the effort expended.

The switch from Catamount to CNL did cause some significant changes, including the use of Firehose and incorporation of thread safety (which we did not elaborate on in this paper). While the gains from implementing Firehose are nothing like the 2-fold improvements seen moving from portable mpi-conduit to native portals-conduit, we still find a 3.5 to 8.5% improvement for transfers larger than the 4K page size to be worth the effort. It was also a continued validation of the Firehose design.

While there is a less than perfect fit between Portals and GASNet addressing schemes, the problem was not the greatest one we faced. By leveraging the restriction that a remote address must lay within the GASNet segment, we were able to use a single statically-created MD and fixed ME to address remote memory. However, supporting the Titanium language would require relaxing this restriction on the remote address. Doing so might also be expected to improve support for the Chapel language. While Firehose is used in portals-conduit only to manage local memory registration, it was originally designed to also deal with the more complex problem of managing remote memory registration. Use of Firehose to its full potential in portals-conduit would allow accessing all of memory both locally and remotely and is potentially the most interesting future work to be done within portals-conduit.

The implementation of Active Messages over Portals was the biggest complication in this port. The use of a locally managed MD, two distinct EQs and credits to manage them eventually all fell into place. The most significant work done in this area is the dynamic credit management to replace static partitioning. This work has significant potential for future generalization to other network conduits in GASNet.

Overall, the GASNet Portals conduit has proven invaluable to a number of PGAS compiler efforts for the Cray XT machines. The Berkeley and Intrepid (gcc-based) UPC compilers both run on GASNet, as do the recent releases of the Cray UPC, CAF and Chapel compilers. The GASNet API has emerged as a common communication layer for these languages, and has enabled new language implementations with a practical model for portability across machine architectures and generations. Finally, the GASNet porting experience may provide useful feedback to the designers of low level communication layers like Portals, demonstrating the use of mechanisms that are valuable in implementing fast one-sided communication.

Acknowledgments

We would like to thank Cray for funding the ports of GASNet and the Berkeley UPC Complier to the Cray XT series, and providing machine access for development and testing. We would particularly like to acknowledge the valuable technical assistance we received from Cray employees Kyle Hubert, Howard Prichard and Doug Gilmore.

This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract Nos. DE-AC02-05CH11231 and DE-FC03-01ER25509.

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.

This research used resources of the Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

This research was supported in part by the National Science Foundation through TeraGrid resources provided by Pittsburgh Supercomputing Center.

About the Authors

Dan Bonachea is finishing up his Ph.D. in Computer Science at the University of California at Berkeley. He is a main developer on the GASNet and Berkeley UPC projects, and his research interests include HPC languages, compilers and runtime/communication systems. Email: bonachea@cs.berkeley.edu

Paul H. Hargrove has been a Principle Investigator in the Future Technologies Group at LBNL since 2000. In addition to working on the Berkeley UPC and GASNet projects, Paul leads the Berkeley Lab Checkpoint Restart (BLCR) project which has recently been ported to the Cray XT. Email: PHHargrove@lbl.gov

Mike Welcome, previously a member of the Future Technologies Group at LBNL, is currently a staff member of the National Energy Research Scientific Computing (NERSC) Center in the Mass Storage Group. Email: mlwelcome@lbl.gov

Katherine Yelick is a Professor of Electrical Engineering and Computer Sciences at the University of California at Berkeley and Director of the National Energy Research Scientific Computing (NERSC) Center at Lawrence Berkeley National Laboratory. She has led or co-led the Berkeley UPC, Titanium, and Bebop projects and her current research interests include parallel computing, memory hierarchy optimizations, programming languages, and compilers. Email: yelick@nersc.gov

All four authors may be reached by paper mail to:

Lawrence Berkeley National Lab
One Cyclotron Rd.
Berkeley, CA 94720

References

1. Dan Bonachea. *GASNet Specification v1.1*. UC Berkeley Computer Science Division Report CSD-02-1207, 2002.
2. *Berkeley Unified Parallel C (UPC) Project.* http://upc.lbl.gov
3. *Titanium Project Home Page.* http://titanium.cs.berkeley.edu
4. *GCC UPC (GCC Unified Parallel C).* http://www.intrepid.com/upc.html
5. *Co-Array FORTRAN at Rice University.* http://www.hipersoft.rice.edu/caf/
6. *Chapel Programming Language Homepage.* http://chapel.cs.washington.edu
7. Cray Inc. *Cray C and C++ Reference Manual*. December, 2008. Publication number S-2179-70. http://docs.cray.com/books/S-2179-70
8. Cray Inc. *Cray FORTRAN Reference Manual*. December, 2008. Publication number S-3901-70. http://docs.cray.com/books/S-3901-70
9. Rolf Riesen, Ron Brightwell, Kevin Pedretti, Arthur B. Maccabe and Trammell Hudson. *The Portals 3.3 Message Passing Interface.* Sandia National Laboratories Report SAND2006-0420, 2006.
10. Cray product information from http://www.cray.com/products
11. Alan Mainwaring and David Culler. *Active Message Applications Programming Interface and Communication Subsystem Organization.* UC Berkeley Computer Science Division Report CSD-96-918, 1995.
12. Christian Bell and Dan Bonachea. "A new DMA registration strategy for pinning-based high performance networks." in *Proceedings 2003 International Parallel and Distributed Processing Symposium (IPDPS 2003).* Nice, France, 2003.
13. *Franklin Home Page.* http://www.nersc.gov/nusers/systems/franklin/
14. Dan Bonachea and Jason Duell. "Problems with using MPI 1.1 and 2.0 as compilation targets" in *2nd Workshop on Hardware/Software Support for High Performance Scientific and Engineering Computing (SHPSEC-03),* 2003.