Title

On the conditions for efficient interoperability with threads

Permalink

https://escholarship.org/uc/item/47q525qm

ISBN

978-1-4503-2642-1

Authors

Ibrahim, Khaled Z

Yelick, Katherine

Publication Date

2014-06-10

DOI

10.1145/2597652.2597657

Peer reviewed

On the Conditions for Efficient Interoperability with Threads: An Experience with PGAS Languages Using Cray Communication Domains

Khaled Z. Ibrahim, Katherine Yelick
Lawrence Berkeley National Laboratory
One Cyclotron Road, Berkeley, CA 94720, USA
{KZIbrahim, Kayelick}@lbl.gov

ABSTRACT

Today's high performance systems are typically built from shared memory nodes connected by a high speed network. That architecture, combined with the trend towards less memory per core, encourages programmers to use a mixture of message passing and multithreaded programming. Unfortunately, the advantages of using threads for in-node programming are hindered by their inability to efficiently communicate between nodes.

In this work, we identify some of the performance problems that arise in such hybrid programming environments and characterize conditions needed to achieve high communication performance for multiple threads: addressability of targets, separability of communication paths, and full direct reachability to targets. Using the GASNet communication layer [6] on the Cray XC30 as our experimental platform, we show how to satisfy these conditions. We also discuss how satisfying these conditions is influenced by the communication abstraction, implementation constraints, and the interconnect messaging capabilities.

To evaluate these ideas, we compare the communication performance of a thread-based node runtime to a process-based runtime. Without our GASNet extensions, thread communication is significantly slower than processes–up to 21× slower. Once the implementation is modified to address each of our conditions, the two runtimes have comparable communication performance. This allows programmers to more easily mix models like OpenMP, CILK, or pthreads with a GASNet-based model like UPC, with the associated performance, convenience and interoperability advantages that come from using threads within a node.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems—Network operating systems; D.1.3 [**Programming Techniques**]: Concurrent Programming—Parallel and Distributed programming

ICS'14, June 10–13 2014, Munich, Germany.

ACM 978-1-4503-2642-1/14/06 ...$15.00.
http://dx.doi.org/10.1145/2597652.2597657.

Keywords

Interoperability, Communication Paradigms, Programming Languages, Processes, Threads

1. INTRODUCTION

As hardware designers continue to take advantage of transistor density increases while addressing power issues, computing systems are growing increasingly complex, with tens of cores per chip and and multiple chips organized into shared or distributed memory systems. HPC platforms have witnessed multiple transitions, with vector machines using mostly shared memory, massively parallel processors using distributed memory, and multicore processors again using shared memory. The multicore era in HPC has brought back shared memory programming within the node, because it is both convenient for programmers and efficient in terms of memory usage and intra-node communication.

Extreme-scale systems currently use hierarchal shared and distributed resources, which leads to multiple programming models. Software abstractions to efficiently deal with resource sharing are typically different from those requiring privatization and protection. Processes provide the natural abstraction for protection, while threads provide the lowest overhead abstraction for sharing. For instance, the Message Passing Interface, MPI uses processes as the default mechanism for parallel execution, while OpenMP is built on top of threads.

In a threaded model, access to shared resources such as network buffers involves mutual exclusion to coordinate access to the resource. Processes simplify access to these resources through privatization and protection, allowing concurrent accesses without worrying about serialization at the user level. On the other hand, processes have disadvantages for parallelism within a node, leading to memory overhead due to replication of data and time overhead due to expensive sharing mechanisms, *i.e.*, mmap files for data sharing or RPC for code sharing. While most node programming models use threads to provide low-overhead shared-memory programming, these threads cannot initiate communication efficiently to other nodes in scalable systems. We measure up to 41× slowdown for transferring small messages by threads compared with processes. These constraints also influence the application tuning efforts and strategies while using hybrid programming.

Many research efforts discussed techniques for reducing the overhead associated with using threads while injecting messages to the interconnect [2, 4, 16, 3, 15]. The main theme of these efforts is to parallelize message injection by reducing the size of critical regions that protect shared resources, including the use of multiple communication endpoints when allowed by the underlying messaging system. This only partly solves the problem because it never fully eradicates serialization. As such, the performance gap between threads and processes continues to increase with the number of cores per node, on all systems we explored.

In this work, we identify the necessary conditions to make threads communicate efficiently enough that they are comparable to process performance. These conditions are targeting *addressability*, *separability*, and *reachability*. The addressability condition requires precisely specifying the target of a transfer before initiating it. The separability condition assures a fully parallel transfer management for independent messages. The reachability condition requires having a direct path to each possible target. We show the details of how to satisfy these conditions to improve a pthread-based implementation of GASNet [6]. Our implementation currently works on Cray Gemini and Aries interconnects, used by multiple machines of the top ten supercomputers in the world [18]. Our technique is conceptually simple but effective. It combines the creation of multiple communication domains with redundant registration for memory segments with interconnect driver to create fully parallel communication paths for threads. Through the performance of microbenchmarks and applications, we show that GASNet can deliver the same performance regardless of the composition of threads and processes chosen by the runtime.

The contributions of this work include: a detailed description and analysis of our work to improve the support for threading in GASNet, and our definition of the necessary conditions to allow threads to communicate efficiently. To the best of our knowledge, our released software is the only publicly available solution on latest Cray supercomputers, where the inter-node communication performance of processes and threads matches. The presented work can easily be implemented on other platforms as well. It also enables efficient integration of PGAS programming languages, such as UPC, with programming models that rely on threading models, for instance, HabaneroC and OpenMP.

The rest of this paper is organized as follows: After describing experimental setup in Section 2, we briefly describe, in Section 3, the motivation of supporting programming models based on processes and threads in modern runtimes and the challenges in achieving efficient interoperability. We present GASNet and our work to achieve efficient interoperability between processes and threads in Section 4. We also layout the necessary conditions to achieve optimal interoperability in general. Performance analysis of our scheme is presented in Section 5. We discuss related work in Section 6, and conclude in Section 7.

2. EXPERIMENTAL SETUP

We used NBP benchmarks written in UPC, which are distributed with Berkeley UPC. We also use a UPC implementation of the UTS benchmark [12]. For micro-benchmarking, we used a modified version of an OSU microbenchark [14] to measure the bidirectional latency of data transfers with concurrent communication.

Most of the presented experiments in this study are carried out on Cray XC30 supercomputer (Edison) [1], installed at NERSC. Edison peak performance is 2.39 petaflops/sec. Each node has two socket Ivy-Bridge processors at 2.4GHz and 64 GB memory. The Edison interconnect (named Aries) has a Dragonfly topology [5]. Aries uses tiled router architecture, where 4 nodes are connected to each router. Traffics from different nodes are multiplexed by the router on a packet-by-packet basis thus allowing nodes to exceed their fair share of the bandwidth. The messaging unit can be programmed using the Generic Network Interface (GNI), and the Distributed Shared Memory Application (DMAPP) APIs. GASNet library, similar to Cray MPI implementation, is developed on top of GNI. The same APIs are used on earlier generation Cray XE06 (Hopper), with the Gemini interconnect. We also conducted microbenchmarking of other runtimes on the IBM BlueGene/Q BGQ (Mira) at Argonne National Laboratory, and the Trestles infiniband cluster at San Diego Supercomputing Center.

3. SCALABLE RUNTIME DESIGNS

Most scalable runtimes, such as MPI [9] and UPC [19], rely on processes in designing their runtimes because they provide a protection mechanism in accessing resources. The operating system provides replicated software data structures to manage shared hardware resources such as the network interface (NIC). This relieves the messaging runtime from using mutual exclusion to access the interconnect. Threads can be used within a compute node to utilize shared memory programming models such as OpenMP, but they suffer long latencies to communicate across nodes due to serialization, as detailed in this section.

3.1 Process-based Runtimes

The namespace (address space) replication mechanism with processes matches well distributed hardware resources, which are dominant in scalable machines. Figure 1 shows the communication domain abstraction used in Cray XE06 and XC30 supercomputers. The job spawner creates and distributes application processes on computational nodes based on the user request. These processes collectively create a communication domain using the information provided by the spawner runtime. Each process creates endpoints, and completion queues through which it can inject transfers, track their completions, or get notifications for incoming messages. Applications typically register part of their memory to allow faster communication. With memory registration, the OS guarantees not to change the mapping between the virtual and physical memory. Depending on the programming model, the registration information can be exchanged at the beginning of the application by the runtime, or on demand. GNI provides two communication mechanisms, a messaging mechanism with mailbox like semantics, and a Remote Direct Memory Access (RDMA) mechanism for one-sided communication. The messaging mechanism allows efficient two-sided small transfers, while the RDMA mechanism delivers higher performance for large transfers.

A single communication domain is typically created by the runtime for an application, except when multiple runtimes are used by the same application, for instance, mixing UPC and MPI or using GNI messaging and DMAPP collectives.

3.2 Thread-based Runtimes

The advent of multicore to node designs makes memory sharing within a node and thus hierarchal designs more common in HPC systems. Threading is an attractive abstrac-

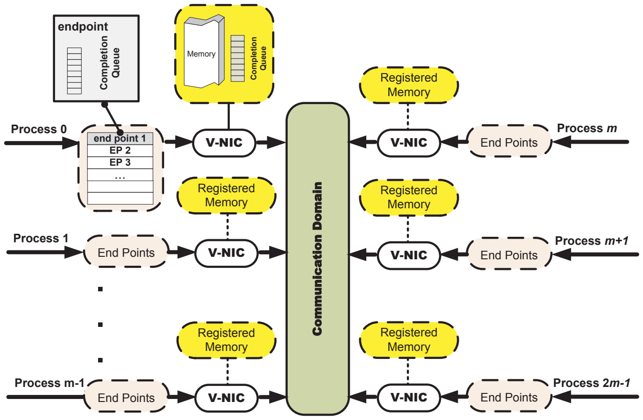


Figure 1: Cray GNI Communication Domain abstraction: each process is assigned a unique access point that can be used without mutual exclusion.

tion for sharing architectural resources because they expose sharing to the application and the runtime using a single namespace (virtual address space). The resource sharing makes the development of runtimes for shared-memory programming models, such as OpenMP, much simpler. It also makes the development of tasking runtimes, such as HebaneroC, more efficient. In these models, the workload is distributed between working threads to achieve load balancing. At the OS level, threads share the page tables, file resources, etc. This means that no protection is provided between executing threads, but a smaller memory footprint is used by the application.

The only limitation for threads is that a correct access to mutable shared resources could require mutual exclusion. Applications and runtimes can use locks, atomics or transactions to enforce serialization. For HPC workloads, one of the most critical shared resources for performance is the network interface. Enforcing serialization for accessing the network leads to a significant performance penalty. For instance, accessing the network massaging endpoint is not thread-safe in Cray GNI, leading to the serialized multiplexing depicted in Figure 2. Other massaging system, such as IBM PAMI, provides a thread-safe access to endpoints [8] when multiple threads do not share the context of an endpoint.

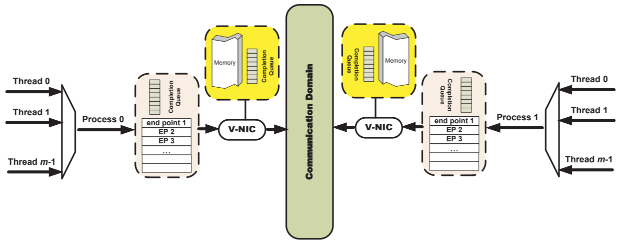


Figure 2: Default support of multithreading through multiplexing threads, with runtime mutual exclusion, into the same interconnect access point.

3.3 Internode Communication Performance of Processes vs. Threads

To quantify the performance of using threads to send messages (or memory transfers) to remote nodes, we show two scalable programming models, MPI for two-sided communication and UPC (based on GASNet) for one-sided communication. Although MPI also provides one-sided communication support, it is not well tuned on all the implementations we explored, including Cray MPI implementation, OpenMPI and MPICH.

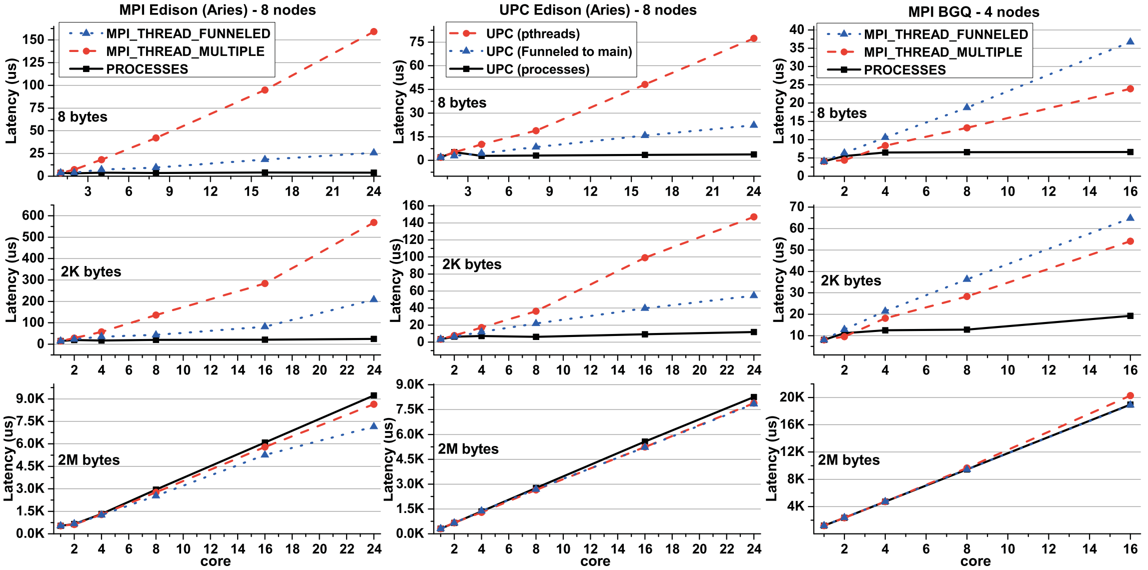
The bidirectional latency microbenchmark is based on OSU benchmarks [14]. We measure the latency by issuing a single non-blocking send and posting a non-blocking receive to the target rank, then waits for the completion of both messages. Ranks are placed such that each pair resides in different nodes. The communication between a pair is bi-directional. We report the average latency over thousands of messages after warming.

When threads are used with MPI, we used different tags to resolve pairing ambiguity. We also use different communicators between threads because some runtimes use a hash of the rank and the communicator to parallelize the injection to the network [8]. Communicators have per rank membership, thus allowing different threads to use different communicator leads to subscription of each rank with all communicators. For threading with MPI, we experimented with three modes: `MPI_THREAD_MULTIPLE`, which allows all threads to inject messages to the interconnect, `MPI_THREAD_FUNNELED` where the main thread injects the messages of all threads, in addition to the default mode of one thread per process. For funneling, we do not account for the extra synchronization to notify the main thread with the readiness for messages of other threads.

We created a UPC version of the same benchmark using one-sided `get` transfers with similar to MPI benchmark pairing of threads. We report for Berkeley UPC, based on GASNet, which supports processes and threads models; whereas Cray UPC supports only processes. Because the shared address space of each UPC thread is semantically similar, whether the runtime (GASNet) uses processes or pthreads, we did not need any additional handling at the application level. GASNet supports three threading modes analogous to MPI [6].

As shown in Figure 3, for 8B messages, processes deliver the lowest latency for data transfers, independent of the concurrency level. This observation is valid for both programming languages, UPC and MPI, and different implementations of MPI. The network can sustain more traffic, thus runtime overhead and serialization dictate the performance. The use of threads increases the latency of transferring messages because most runtimes serialize the access to their data structure using locks, atomics, or transactions. For 8B messages at concurrency level of 24, Figure 3.a&b, the latency increase for MPI threads over processes by 41×, while for UPC the increase is 21×. The gap increases with the level of concurrency, which is an alarming trend because future systems are expected to have more cores. The difference decreases as the message size increases because the performance becomes bounded by the available bandwidth. For 2KB messages using UPC, the latency ratio for threads to processes gets reduced to 12× and approaches 1× for 2MB messages. Using funneling to the main thread can lead to better performance than parallel injection by all threads.

We found similar performance trends, to those shown in Figure 3, on other runtimes including OpenMPI [13], MPICH [10], MVAPICH [11] on different interconnects including Cray Gemini and Infiniband clusters. In some runtimes, such as



(a) (b) (c)

Figure 3: bidirectional Latency microbenchmark of UPC and MPI programming models for processes vs. two threading models (THREAD_MULTIPLE for full threaded injection to the interconnect, and THREAD_FUNNELED for relaying communication to the main thread).

OpenMPI, the high performance byte transport layer is disabled when threading is enabled. This leads to a larger performance gap between threads and processes. As such, many scalable runtimes disable threading support by default. We measured up to 366× latency increase for threads compared with processes for MPICH implementation on Infiniband cluster (Trestles) at a concurrency level of 32. MVAPICH has up to 96× latency increase on the same cluster.

In the systems we explored, the best runtime in handling this problem is IBM MPI implementation on BGQ systems, shown in Figure 3.c. The latency difference between processes and threads is at most 4× for 16-way concurrency. BGQ systems support L2 atomic locks that lower the serialization overhead. These efficient locks are suitable for single socket nodes, which is a distinct path for architecting nodes that is not prevalent. IBM MPI implementation uses PAMI endpoints and context abstractions, which could allow fully concurrent injection of messages to the interconnect. Processing incoming messages with threads cannot be fully parallelized [8], though.

4. IMPROVING GASNET THREADING SUPPORT USING MULTIPLE COMMUNICATION DOMAINS

We argue for a runtime to support full concurrent communication by an execution abstraction, such as threads, it needs to satisfy three conditions: unambiguous *addressability* of remote targets, *separability* of communication paths, and full direct *reachability* between all communication parties. In this section, we discuss the challenges associated with these conditions and the way we handled them in GASNet [6], Global Address Space Networking, to improve one-sided communication primitives.

4.1 GASNet Runtime and Base Threading Support

GASNet is a scalable communication library that provides, at the core of its functionality, APIs supporting active messages (AMs). Memory segments are typically registered for communication between ranks of an application. Likewise, AM handler functions are registered with the library. An AM carries both a payload and a handler *id* of the task to be executed at the remote side. AMs give unique ids only to processes (calling them nodes).

GASNet provides another set of communication primitives for one-sided memory transfers and collectives, called extended APIs, which does not require any remote side processing of the data. These primitives typically exploit network accelerated remote memory access (RMA) mechanisms.

GASNet library is used by multiple parallel partitioned global address space (PGAS) programming languages such as UPC, Titanium, and Co-Array Fortran. Each of these languages uses a different subset of GASNet functionality. The library supports pthreads in three modes resembling MPI's support, where the language runtime should declare if one or more threads need to concurrently call GASNet. If threading support is requested, GASNet enforces serialization of accesses to shared runtime resources through mutual exclusion (using locks or atomics). GASNet provides different builds depending on the level of threading support. This allows removing unnecessary handling for thread safety if only one thread per process is used.

4.2 One-sided Primitives and Threading

One-sided communication typically involves transfer between a local memory and a remote memory, using either `put` or `get` primitives. The remote memory can either reside locally (within a node) or at a remote node. Most interconnect HPC systems provide a hardware acceleration mechanism for one-sided primitives. Although these APIs are not the core APIs for GASNet, they are the most critical to performance for many high-level programming languages, such as UPC. In this section, we focus on how to improve the support of threaded one-sided communication in GASNet.

4.2.1 Addressability of the Remote Destination

One-sided communication in GASNet uses a tuple of a registered address and process *id* for the remote part of the transfer. The process *id* is used solely to resolve the affinity of the memory address to a particular compute node. The affinity is enough for GASNet runtime to manage the transfer, and the participation of the remote process is typically not needed. PGAS languages, such as UPC, use these tuples to create a global unique name for each shared memory address. One-sided communication in GASNet does not care about which entity is going to operate on the data at the remote location. Thus, the use of threads within a process does not pose any address ambiguity challenge.

4.2.2 Separability of Communication Paths

Separability of communication paths is the condition where independent transfers do not get serialized unnecessarily by the runtime. Full separability requires special handling during data transfer injection, progress and advancement, and reception. Separability of communication paths necessitates carful communication resource allocation and management.

Communication resources include the runtime data structures used in holding the communication state. Communication management involves mechanisms and state machines used in initiating communication, advancing the progress, and checking for completions. We argue that to support threads efficiently we need full separability of resources and management by each thread.

To achieve separability of resources at the GASNet runtime level, we use exclusive per thread resource pools. This alleviates the need for locks or atomics in the case of having shared resources. Examples of these resources are communication handles and descriptors, and internal bounce buffers. Likewise, we ensured that all used libraries do not use any shared resources. For instance, most memory allocation libraries use the shared heap to allocate memory, thus causing serialization. We made sure that the memory allocator uses a distinct heap per thread.

The challenge we faced is that the interface of the Cray GNI library is not thread-safe. Assigning a distinct endpoint to a thread is not enough to achieve concurrent injection to the interconnect. In this work, we solve this problem by creating a separate communication domain for each thread (or group of threads). When each thread is assigned a separate domain, as shown in Figure 4, all threads can concurrently inject to the GNI layer without any serialization.

4.2.3 Full Direct Reachability to All Remote Targets

The challenge with the creation of multiple communication domains is that it makes it potentially difficult to have full reachability to all memory addresses. Using multiple domains, each thread subscribes to a different communication domain. Registering the memory of affinity to a particular thread leads to unreachable destination memory segments because each thread can only see the memory of threads subscribing to its domain. The shared memory is typically split between executing units. A shared memory segment always has affinity to one execution unit, which is responsible for registering this segment of the shared memory with the messaging runtime. It is also responsible for exchanging information about these segments. In the base implementation, only the main thread does the registration of the whole memory assigned to the process. Multiple processes sharing a node register their segments independently and then exchange information. Thus, each memory segment is registered once with the interconnect.

To solve this problem, we register the whole memory of affinity to the process that this thread belongs to. This leads to redundant registrations of the same memory depending on the number of communication domains created by the runtime. These redundant registrations (aliasing), shown in Figure 4, allow full direct reachability for each thread to the whole shared address space. In fact, this allows multipath reachability to each memory location. Fortunately, this does not affect the consistency model provided by GASNet because GASNet always guarantees remote completion of `put` operations. The implication of redundant registration on the registration resources is discussed in Section 5.2.

4.3 Active Messages and Threading

GASNet active message APIs allow sending a request to execute a handler procedure at a remote destination. The request carries the data, the destination and the handler *id*. It supports a strict request reply mechanism, thus the destination can send at most one reply to the sender. The target *id* in a reply is implicit (the sender of the request). GASNet AMs can be used in many management tasks by high-level programming languages, for instance, to implement synchronization primitives and collectives.

4.3.1 Addressability of the Remote Destination

The destination of an active message, similar to MPI, is only a process, and the handler can be executed by any thread belonging to the process. Unlike one-sided primitives, the destination process strictly specify the partner responsible for processing the message, not just the affinity of the destination memory. The specification of processes as the only valid target arose when most HPC node architectures were a single core, or processes were thought as the main scalable runtime abstraction. GASNet AMs do not recognize threads as addressable entities. Allowing the active message handler to be executed by any thread belonging to the target process can be looked at as a flexibility because it allows low-loaded threads to execute the handler. If multiple communication domains are used, this causes complexity in runtime design to maintain correct execution as discussed in the next sections.

4.3.2 Separability of Communication Paths

Active message (AM) requests and replies require explicit resource allocation at the sender side. The receive side resources is transparently managed by the runtime. This makes concurrent injection of AMs by different threads an easy task using multiple communication domains. The receiver of an AM does not post any explicit receive, thus resource management at the receive side is completely controlled by the runtime. Separability of resources management is complicated because processing an incoming AM should not be done by multiple threads. Thus, the reception cannot be fully parallelized and a centralized decision with a mutual exclusion mechanism needs to be used.

4.3.3 Full Direct Reachability to All Remote Targets

The reliance of active messages (AM) on addressing processes makes direct reachability achievable with one communication domain even with multithreading. The reachability of AMs is dependent on not only the arrival of the data to the destination but also on the execution of the handler by the target. The data reachability for large messages is achieved using multiple domains, using the one-sided RDMA. The handler information are sent using small messages, where most vendors provide a mechanism for efficient mailbox short messages on pre-allocated buffers. The messaging mechanism is solely used for the data and the AM descriptor when the payload in small.

The GASNet AM specification of the handler execution imposes the following runtime behavior: any thread trying to make progress should check for arrival of AMs because of the possibility that this thread is the only one doing so. If multiple paths of arrival are possible, all paths should be checked for incoming messages. On the other hand, if multiple threads are ready to execute a handler only one should do so. Thus, the use of multiple domains for AMs creates the possibility for all threads advancing all domains. Each advancement requires a mutually exclusive access of the communication domain. Therefore, having multiple reachability paths can lead to a significant serialization problem.

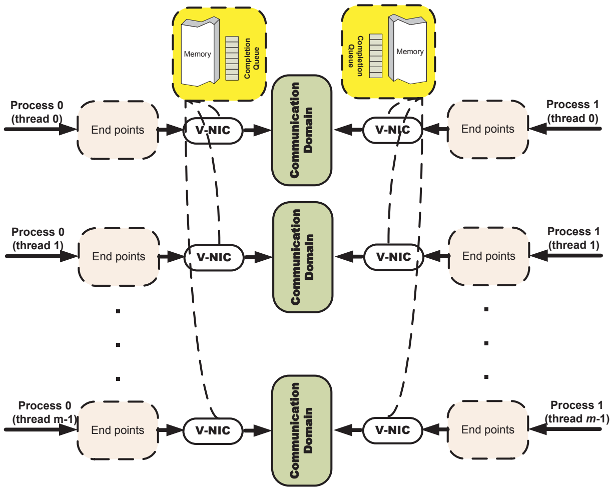


Figure 4: Multipath through the use of multiple communication domains. Each thread can subscribe to a distinct domain. The same address space is redundantly registered in multiple domains.

If all threads are addressable, then we will need for direct reachability communication domain count equals the square of thread concurrency. The resources needed for a communication domain make the number allocatable of domains limited, as detailed in Section 5.2.

In our implementation, we made the choice of restricting AM handler processing to the first domain to minimize the potential serialization. The data transfer part can go through any domain, especially for large transfers. The first domain is by default accessible to the first thread of a process. Given that some languages using GASNet may use other than the first thread for receiving and processing AMs, we make all threads infrequently advance the progress of the first domain AMs. Only the first thread within a process has a low latency in processing the AM reception. We use an environment variable to control how frequently each thread should check and advance the AM domain.

5. ANALYSIS OF THE USE OF MULTIPLE COMMUNICATION DOMAINS

In this section, we show that the use of multiple communication domains improves the performance of threaded one-sided communication dramatically. We also show the impact on the performance of active messages. We finally discuss the resource issues associated with the use of multiple communication domains.

5.1 Improvement of the performance of one-sided primitives

In Figure 5, we show that the performance of a one-sided `get`, with different levels of thread concurrency and domain count. We show the latency for small to medium message sizes on Cray XC30 (Edison). We plot performance of threads relative to using pure processes for communication, for the microbenchmark presented in Section 3.3. The first observation is that using domain count matching the number of threads yields performance equivalent to process-based implementation (at most 8% difference for small messages) due to infrequent handling active messages. Without active messages, the performance perfectly matches, and even yields a slightly better performance for threads.

The improvement is larger for high thread concurrency and small messages. The improvement for threads performance is up to 31.5× for 8B messages with 24-thread concurrency. The improvement decreases with the increase of the message size because the bottleneck shifts to bandwidth availability. For 2KB messages the improvement is up to 14.6×. The largest message we observed improvement for is 128KB. In earlier generation Cray Machines, XE 06 (Hopper), the largest message to see benefit is 32KB, smaller than that for XC30 (Edison).

The second observation is that using a domain count less than the thread count significantly reduces the latency of injection. The latency is reduced monotonically with the number of domains. Thus depending on the resource constraints of allocating domains, increasing the domain count can bring the internode communication of the threads closer to processes.

5.2 Resources Allocation

Multiple resource constraints affect the approach presented in this work: the maximum domain count allowed by the messaging library, and the available memory registration resource, and memory allocation per process. In this section, we discuss the implication of these resource limits.

Most interconnect runtimes impose a restriction on the number of domains that can be allocated. On Cray XE06 (Hopper), we can allocate at most 30 domains. On Cray XC 30 (Edison), the number of domains is at most 120. On other architectures such as IBM BGQ, PAMI allows 48-64 communication contexts. The impact of this restriction is that we cannot have communication domains equal to the square of the thread concurrency, up to 64 in most recent supercomputers. This makes direct reachability using separate domains not feasible for active messages. One-sided

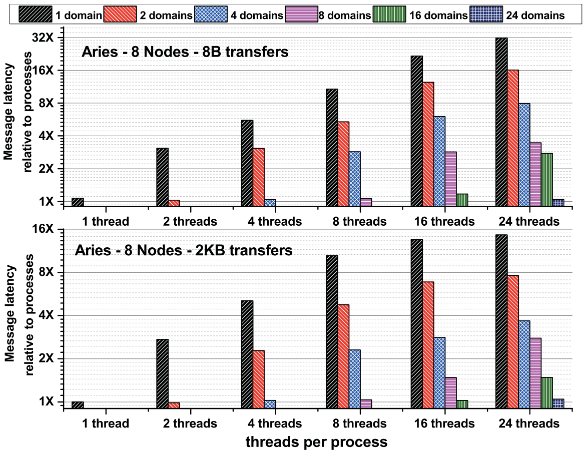


Figure 5: Improvement of message latency with the number of domains for small to medium message sizes on Cray Edison (Aries interconnect).

communication requires domain count at most equal to the number of cores.

A more challenging restriction is the limited registration resources of the memory with the NIC. Some interconnects, such as Gemini (Cray XE06), have limited hardware resources for storing the registration information. This limits the number of memory pages registered per domain. Having redundant registration stress the limited centralized resource. Accelerated RDMA mechanisms rely heavily on memory registration. Fortunately, the registration resources depend mostly on the page count, not the page size. As such, the use of huge pages allows allocating larger registerable memory per thread. In Table 1, we show that the registration resources decline with the number of domains, especially with small pages (4KB), to reach only 64MB per thread when 24 communication domains are used. Using huge pages significantly alleviates this restriction and push the limit to 4GB. Fortunately, for the newer generation Cray XC30, this restriction is no longer an issue, and the registration resources are designed such that it can hold the maximum allocatable memory per process for any number of communication domains.

We also observed a restriction that the maximum allocatable memory, by libc, for a process is 8GB, which is smaller than the available physical memory. It is possible to allocate larger memory using posix allocators. The consequence of the last two restrictions is that we may need to use multiple processes to achieve optimal resource allocation. For instance, we allocate one process per NUMA node and use threads to exploit all cores within that node.

Table 1: Cray Hopper max. registration per thread

Domain Count	small pages	huge pages (8MB)
1	512MB	8GB
2	512MB	8GB
4	256MB	8GB
8	128MB	4GB
24	64MB	4GB

Another comment is that even though the requested network resources are increased with the creation of multiple communication domains, the use of threads usually reduces the physical memory used by the application compared with processes. The saving for threads comes from sharing code segments, and using a single page table per process (shared by all threads). Applications developers also do not use data replication in thread-based programming model, which is typical with processes in distributed programming languages. The runtimes developed over threads are also known to have a smaller memory footprint.

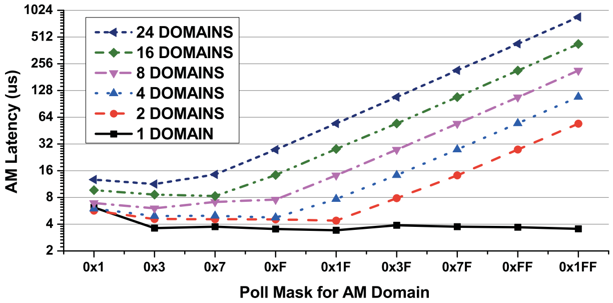


Figure 6: Impact of the choice of polling masks on the latency of GASNet Active Messages.

5.3 Impact on Active Message Performance

As discussed earlier, to avoid frequent locking of communication domains by all threads, we make active message (AM) descriptors go through the first communication domain. To guarantee forward progress, we allow all threads to infrequently advance the AM engine using a mask value[1]. The larger the mask, the less frequent advancement to the AM domain. This leads to the trends shown in Figure 6 by TestAM benchmark[2]. The latency can increase by up to 76×, when the polling mask changes from 1 to 511. Obviously the difference gets smaller as we reduce the number of domains. These latency values are the max for all threads. Note, the performance of the first thread is typically not impacted by the use of multiple domains or the polling mask.

Figure 7 shows the impact on one-sided transfer latencies when the mask is changed. We notice that the mask value of one increases the latency of messages by at most 2.73× for the smallest messages. The difference decreases with the increase in message size until it becomes neutral for messages above 128K. The other observation is that with the smallest mask (highest frequency of polling), the latency is much smaller than using a single domain, shown in Figure 3, by up to 7.3×.

The trends shown in Figures 6 & 7 may suggest that to balance the performance of one-sided primitives with AMs we need to choose a mask value of 0x7. We do not follow that because we know that most languages using GASNet rely more on the one-sided primitives performance. Consequently, we made the default value 0x1FF. To alleviate the impact on AMs, we modify the use pattern of upper layer runtimes to GASNet. For instance, unified parallel C (UPC) runtime is modified such that synchronizations that use GASNet always elect the first thread to receive AMs. The default behavior is such that threads within a process

[1] The mask value is set using the environment variable `GASNET_AM_DOMAIN_POLL_MASK`.

[2] TestAM is a benchmark distributed with GASNet.

elect a thread (normally the last arrival) to perform AMs in behalf of the process. We will show that this strategy proves suitable for applications written in UPC in the following section.

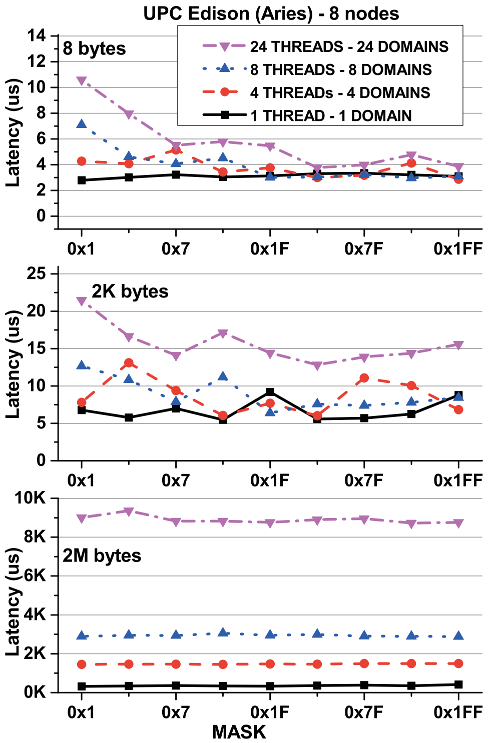


Figure 7: Impact of choice of polling masks on the latency of get operations.

5.4 UPC Application Performance with Communication Domains

To test the efficiency of the presented scheme, we use multiple UPC applications from NBP benchmark. Our objective is to measure the impact of using programming models relying on processes with those relying on threads. Fortunately, Berkeley UPC can run with single threaded processes, or with multi-threading. A UPC thread is mapped either to a process or a thread. Berkeley UPC uses GASNet communication library to achieve portability. All UPC threads (processes or pthreads) initiate communication (no funneling). The presented applications mostly rely one-sided primitives for communication. They also use synchronization primitives and collectives that use AMs. We use 16 UPC threads per node and 16 nodes to make it easier to create power of two, or square processes needed to run these benchmarks. We varied the mapping of UPC threads within a node from using two processes per node and 8 threads per process to mapping all UPC threads to processes. We did not extend our run to one process (16 pthreads) per node because Cray XC30 has two NUMA domains per node. All the studied applications do not have explicit NUMA locality control and system utilities on Cray machines allow controlling NUMA allocation only at the process level. All the runs are relative to process-based implementation, which use xpmem shared memory, between processes for intra-node communication.

The base performance is that associated with one communication domain. As shown in Figure 8, the performance degrades as we use more threads with a single communication domain. For CG, the slowdown can reach up to 5.5× for the use of 8 threads per process. For other applications, the slowdown is at most 61%. The amount of slowdown depends on the message sizes used by the application. The smaller the message sizes the higher the performance slowdown. As shown Figure 8, as we increase the number of communication domains the performance improves monotonically for all applications. The difference between processes and threads implementations becomes bounded to less than 10%. The processing of active messages is a contributing factor to the small mismatch between processes and threads implementations. Our choice of the mask for frequency of polling AMs proves efficient for overall performance of the studied applications. Although, we modified the UPC runtime to make AM calls done by the first thread, we still need for compliance with GASNet specifications do infrequent advancement from all threads to the active message domain. As per GASNet specifications, in a threaded mode any thread should be able to advance the progress of the runtime. This infrequent polls cause a small serialization overhead. We should note that allowing highly concurrent access could reduce the performance because of congestion.

5.5 Remarks on Efficient Interoperability Conditions

Satisfying the presented conditions for concurrent wait-free communication are influenced by multiple factors: the specifications by the programming model, the runtime design constraints and implementation strategy, and the capabilities of the messaging systems of the interconnect.

The specifications can influence the addressability of the target with different execution abstractions. We showed that communication primitives in one-sided models do not suffer any addressability issue whether we use processes or threads, while AMs in GASNet suffer an addressability problem for reasons embedded in their specifications. Addressability can be tackled by either having a convention of using communication resources or by amending the specifications. We argue that long-term solution should consider modifying the specification, for the convention might be difficult to follow or to enforce.

Separability of paths and full reachability are typically opposing forces. One can have separable resources and management on current programming models that lead to limited reachability. Berkeley UPC provides teams, conceptually similar to MPI communicators, which allow independent progress within each team. Participation in a single team can lead to a reachability problem to some targets, and dynamic change of teams can lead to a separability problem. Separability is also influenced by the language specifications. If a language imposes certain ordering semantic for thread execution, then this limits the separability while processing transfers.

What eased the integration of this work to GASNet is that it was done transparently with respect to its legacy interfaces and specifications. For instance, although multiple domains and reachability paths are introduced, the completion semantic and ordering guarantees are not modified. Part of this ease came from the fact that addressability in

PGAS languages does not rely on the execution abstraction (processes or threads).

This work shows encouraging results for the integration projects of UPC with other programming models that rely on threading. These results guarantee threads to be able to communicate without serialization by the runtime.

This will not only simplify runtime integration, but will also simplify application development activities. Application developers, being aware of inefficiency of initiating communication from threads, add additional code for preprocessing and postprocessing messages. A main thread typically collects partial results from all compute threads and distributes incoming messages, which involves unnecessary synchronization. Efficient communication by threads can make such practice obsolete.

The importance of this work is likely to increase in future systems because the number of cores per node is increasing. The trends for memory size growth does not show them coping up. Most applications are also likely to run in strong-scaling regime, forcing them to rely on the performance of small messages at high concurrency.

6. RELATED WORK

Interoperability of scalable runtimes, such as MPI and UPC, with shared memory programming has become critically important with the advent of multi and manycore designs to node architectures. The support of thread-based programming models, in scalable communication runtimes, is the subject of many research proposals and prototypes [2, 4, 16, 3, 15]. Most of these proposals target two-sided MPI programming model. MPI, up to 3.0 specifications, deals with threads as non-addressable entities [9, 7], and requires the ability to reason a serial order of concurrent pthreads execution.

The first issue these proposals tried to address is making threads addressable. They propose assigning threads rank *ids* [17], or using an endpoint per thread [8, 16, 3]. While mapping ranks to threads conflicts with the MPI specification, most endpoint proposals try to use some convention to resolve the mapping between threads and the communication resources. For instance, they associate each thread with a unique endpoint that can be mapped to a communication context within the communication rank.

These proposals assume thread-safety of accessing endpoints, which is true on IBM PAMI [8, 16] if each context is associated with a unique endpoint. Allowing low-overhead injection of messages does not guarantee the creation of fully separable communication paths. Separability of managing the communication resources is typically challenging at the receive side. Contexts are typically collectively advanced, which can be done by a communication thread. The cost of advancing all contexts is typically small in IBM BGQ because locking relies on a low-overhead L2 atomic, which is suitable only for single-socket node designs. Kumar et al [8] details why it is tricky to parallelize MPI_IRecv and MPI_WaitAll with threads, even with the use of PAMI endpoints because of the specification constraints of MPI.

The other approach to address thread support is to reduce the runtime overhead of implementing mutual exclusion. Proposals of fine-grained locking or lock-free atomics show promising results in reducing the impact of serialization [2]. Their objective is to make serialization event very brief. Fine-grained per object locking requires a special care to preserve single ordering in locking or deadlocks become a possibility. In large scale runtimes, this simple requirement can be a challenge. This approach looks at the serialization within the interconnect driver as an orthogonal issue. We note that the base GASNet [6] implementation uses lock-free data structures (manipulated with atomics) and still suffer a large penalty as we scale the number of cores especially for small messages. The overhead in serialized access to a non thread-safe messaging APIs (such as Cray GNI) was their main performance bottleneck.

This work introduces a comprehensive analysis for this problem and provides a solution suitable for PGAS languages, which rely one-sided primitives. Our work is publicly released thus allowing other runtime designers to experiment with it, especially those targeting one-sided abstractions.

7. CONCLUSIONS

Hybrid shared and distributed memory are becoming the standard for massively parallel machines. While threads have both a lower memory footprint and some performance advantages for intra-node programming, we have shown that they often exhibit significant performance problems in interconnect communication. This penalty was as high as 21× for UPC and 41× for MPI in our measurements. We present the necessary conditions for efficient interoperability of process-based scalable programming languages with thread-based node models. We implemented this in the context of an extension to GASNet communication library. The first condition, addressability, is found orthogonal to the execution abstraction, processes or threads, for one-sided communication. The second condition, separability, requires restructuring the runtime to avoid having shared resources between threads and also the creation of multiple communication domains on top of the messaging library. The third condition, reachability, is addressed by using redundant registration (aliasing) of the shared memory segments.

GASNet active messages are shown bounded by their limitation of restricting the addressability to processes leading to a difficulty in achieving full parallelization of their transfers with threads. Overall, our approach significantly improves performance of inter-node thread-based communication, allowing it to match the performance of processes in microbenchmarks. We also compared these in an application setting and measured up to 5× performance improvement for NBP benchmarks build with a hybrid implementation using processes and threads. Aside from improving the performance of GASNet-based programming languages, our analysis also identifies the key features that lower level network APIs and hardware need to support for good hybrid performance. We believe these design principles will be increasingly important as the number of cores per node continues to grow.

Acknowledgments

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, and resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported under contract DE-AC0206CH11357.

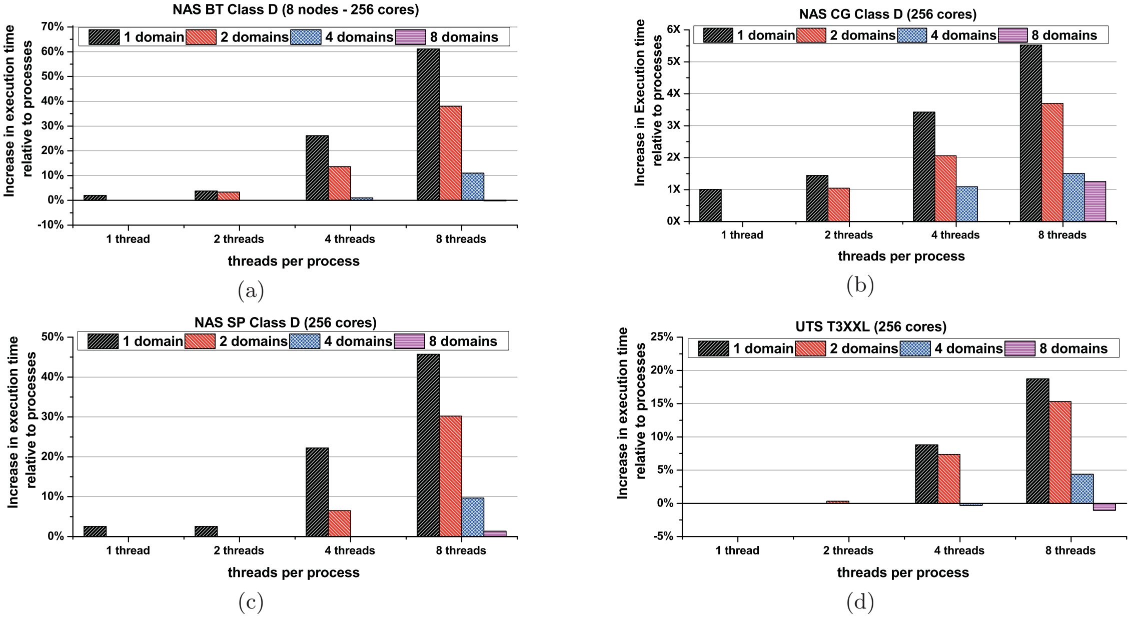


Figure 8: Performance of UTS and NPB applications using different thread concurrency levels and communication domains. All runs use 16 UPC threads per node and 16 Cray XC30 (Edison) nodes.

8. REFERENCES

[1] National Energy Research Scientific Computing Center, Edison Supercomputer. http://www.nersc.gov/users/computational-systems/-edison/configuration.

[2] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. Fine-grained multithreading support for hybrid threaded mpi programming. *IJHPCA*, 24(1):49–57, 2010.

[3] J. Dinan, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur. Enabling MPI Interoperability Through Flexible Communication Endpoints. *EuroMPI*, pages 13–18, 2013.

[4] G. Dozsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman, and R. Thakur. Enabling concurrent multithreaded mpi communication on multicore petascale systems. *Recent Advances in the Message Passing Interface, Lecture Notes in Computer Science*, 6305:11–20, 2010.

[5] 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*, pages 103:1–103:9, 2012.

[6] Global-Address Space Networking (GASNet). Specification v1.8. http://gasnet.lbl.gov.

[7] W. Gropp and R. Thakur. Thread-safety in an MPI implementation: Requirements and analysis. *Parallel Computing*, 33(9):595 – 604, 2007.

[8] S. Kumar, A. Mamidala, D. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burrow. PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer. *The 26th IEEE International Parallel Distributed Processing Symposium (IPDPS)*, pages 763–773, 2012.

[9] 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.

[10] MPICH2 v 3.0.4. http://www.mcs.anl.gov/research/projects/mpich2/ .

[11] MVAPICH2 v 2.0b. http://mvapich.cse.ohio-state.-edu/overview/mvapich2/.

[12] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. UTS: An Unbalanced Tree Search Benchmark. *The 19th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC 2006)*, Nov. 2006.

[13] OpenMPI v 1.7.3. http://www.open-mpi.org.

[14] OSU benchmarks. OMB 4.2. Network-Based Computing Laboratory, Ohio State University, http://mvapich.cse.ohio-state.edu/benchmarks/.

[15] G. Saxena. Thread safety for hybrid programming in thread-as-rank model. Master's thesis, The University of Edinburgh, Aug. 2013.

[16] G. Tanase, G. Almasi, H. Xue, and C. Archer. Network endpoints for clusters of smps. *The 24th IEEE International Symposium onComputer Architecture and High Performance Computing (SBAC-PAD)*, pages 27–34, 2012.

[17] H. Tang and T. Yang. Optimizing threaded mpi execution on smp clusters. *The 15th International Conference on Supercomputing*, pages 381–392, 2001.

[18] Top 500 Supercomputers. http://www.top500.org.

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