Efficient Interoperability of OpenSHMEM on Multicore Architectures

Khaled Z. Ibrahim
Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA
kzibrahim@lbl.gov

ABSTRACT

Most HPC programming models face an interoperability challenge because of the advent of multi/many core architectures [1–3]. Efficient interoperability–for instance, with shared memory programming models such as OpenMP–requires reconsidering the design of various levels of the programming model software stack. While support for interoperability typically exists at the hardware and system messaging library levels, most programming models lack the interfaces that ease such interoperability. In this paper, we discuss requirements of efficient interoperability and show the alternative paths for satisfying them for OpenSHMEM. We discuss the implication of maintaining the current interfaces and enhancements to ease interoperability.

1. INTRODUCTION

Architectural trends show increase in core concurrency in node designs, as well as increase in heterogeneity. Multiple programming models are likely needed to efficiently exploit such architectures. Among metrics for efficient interoperability is the ability to support shared memory models within a node using `pthreads` and distributed models using processes across nodes without performance penalty. For instance, we need runtimes capable of initiating communication within multithreaded parallel region without increase in latency compared with the single-threaded case. This eases programming, reduces synchronization, and allows better scaling.

Most optimized parallel libraries within a node (or a coherence domain) rely solely on `pthreads` model because it utilizes a single name space for data and functions, thus reducing runtime overheads in address and function shipping across compute threads. Shared memory models also allow more efficient memory usage. Physical memory is typically a scarce resource at scale, which could be stressed by communication runtime internal buffers that scale with the run size. Unfortunately, architectural trends show reduction in physical memory per core. At the application level, explicit shared memory programming allows efficient resource allocation (no data replication). While it is possible to support sharing with processes, sharing data requires memory mapping to a shared file (thus creating aliases and offset-based indexing) and sharing functions requires an RPC mechanism. These mechanisms introduce overheads for shipping data or functions. Additionally, processes do not provide a guarantee of physical sharing, thus forcing runtimes to frequently check for sharing before applying optimizations.

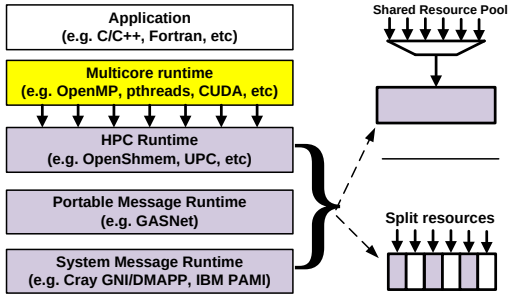

Figure 1: A typical software stack or PGAS language. With the use of `pthreads`*-based runtime, we have the choice of either splitting resources or sharing them and protecting access using a mutual exclusion mechanism.*

2. EFFICIENT INTEROPERABILITY CONDITIONS

Multiple considerations, regarding resource management and addressability of `pthreads`, should be considered across the whole software stack to achieve interoperability. A basic requirement is the need to split the resources at various levels of the software stack such that accessing the interconnect does not involve any serialization (including using locking or lock-free atomic-based algorithms). Effectively, we need thread-specific separate paths in carrying transfers while maintaining full reachability and addressability to the shared state.

In Figure 1, we show a typical PGAS language software stack. The efficiency of the system relies on the aggregate support in all layers. Creating a separate injection path for each thread requires resource split in all layers of the stack. Some system libraries, such as IBM PAMI, support thread-specific allocation of resources, called contexts. Others, such as Cray GNI, are not thread-safe, or thread-safe using library locks, such as Cray DMAPP or Infiniband. Our recent study [3] shows how to use Cray GNI domains, intended to support multiple client runtimes, to create thread-specific separate injection paths. At the second level of the stack (from bottom), earlier approaches for GASNet and MPI [1] argue for lock-free data structures (using atomic operation) to make serialization brief, we found that insufficient. Moving the interconnect controller on-chip makes the injection overhead as small as few tens of nanoseconds. The increase in the number of cores makes any serialization and lock migration extremely expensive. This leads to an increased gap between serialization-free accesses to the interconnect and serialized ones.

In Figure 2, we show the increase in latency for a `pthreads`-based implementation as we increase the level of concurrency

Table 1: Suggested Modification to OpenSHMEM APIs

API [a]	functionality
start_pes(IN npes, INOUT endpoint_count)	Initialize, and request maximum thread-safe resources
shmem_my_endpoint(OUT endpoint)	query default endpoint
shmem_create_endpoint(OUT endpoint)	create an endpoint
shmem_int_put(IN endpoint, IN target, IN source, IN nelems, IN pe)	put operation of integer with injection resource argument
other APIs similar to shmem_int_put	...

[a]Suggested modification from current standard are colored in blue.

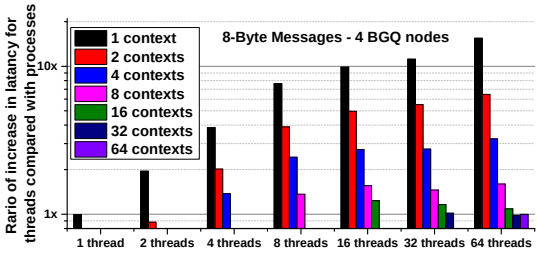

Figure 2: The latency of injecting a transfer using `pthreads` *compared with processes increases with the level of concurrency on IBM BGQ systems. Assigning threads different contexts reduce injection latency.*

on IBM BGQ systems. The latency can be reduced if we attach each thread to a unique PAMI context. BGQ systems provide the lowest possible impact, despite being high, on transfer injections because a node has a single socket, relatively slow compute cores, and a special L2 atomic support that reduces lock migration overheads. Similar trends were observed on Cray XE&XC systems [3]. The impact is typically large for small to medium size messages (affecting up to 128KB transfers on Cray XC nodes compared with 2KB on IBM BGQ nodes). Berkeley UPC/GASNet recently supported a full resource split on all layers of the stack using Cray GNI domains to achieve near optimal one-sided transfers with `pthreads`-based implementation.

While splitting resources allows for concurrent injection, the performance of `pthreads` cannot match processes unless language semantic allows addressability of the target and direct reachability [3]. MPI two-sided transfers [4] and GASNet active messages [3] are two examples not satisfying these conditions. The new specification of GASNet is considering such challenges in its design.

3. EFFICIENT INTEROPERABILITY FOR OPENSHMEM

OpenSHMEM, being part of the PGAS family, has the property that the target of a transfer is independent of the use of thread or process abstractions. The property is beneficial as long as the passive target model is used, *e.g.* using hardware accelerated RDMA. If active target model is used, for instance using a service thread to process incoming transfers, the limitations experienced by MPI [4] and GASNet Active Messages [3] are likely to negatively impact performance. Active target models typically create another bottleneck at the receive end. First, in average, the number of service threads per node at the target needs to match the number of threads injecting traffic, thus consuming compute resources in servicing communication. Second, the service target needs to be explicitly addressed. Otherwise, multiple service threads could contend while trying to load-balance the service for incoming traffic. Overall, an active-target implementation will be far more challenging in supporting efficient interoperability.

As it is now, for an OpenSHMEM runtime to exploit resource split at lower layers, it is required to lookup thread specific identifier of the resources in each call. This operation typically takes several hundreds of cycles on modern architectures, thus impacting small transfers. The problem can be handled transparently using some preprocessing to add the additional thread-specific arguments. GASNet employs such technique, but in future specifications this identifier is added explicitly in the API.

We argue that OpenSHMEM better considers an addition of a resource argument to its interfaces. In Table 1, we briefly outline the proposed modifications. The application can request multiple injection resources `endpoints` that are lock-free. The runtime returns the count of `endpoints` that could be created, the minimum of what is available and what is requested. The application, then, allocates `endpoints` as needed. In multithreaded setting, the memory transfer APIs pass an additional argument to select the resources used for injection. User assertion can be added, whether the user will guarantee thread safety of the endpoint or the runtime.

This resource argument, `endpoint`, is originally proposed for MPI two-sided APIs [2], but the potential for success in PGAS language setting is greater. Hardware accelerated RDMA, makes the only resources that need explicit management the injection resources at the initiator of the transfer. Among objectives of this paper is to reopen the discussion of the implication of adopting such interfaces to both the runtime design and the application layers.

4. CONCLUSIONS

Efficient interoperability is critical to programming models in future systems. Trends of hybrid designs, increased core count, and reduced memory per core make interoperability inevitable. In this paper, we discuss general conditions to achieve efficient interoperability and specific OpenSHMEM considerations to achieve them.

References

[1] 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.

[2] 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.

[3] K. Z. Ibrahim and K. Yelick. On the Conditions for Efficient Interoperability with Threads: An Experience with PGAS Languages Using Cray Communication Domains. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 23–32, 2014.

[4] 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.