OSPRI: An Optimized One-Sided Communication Runtime for Leadership-Class Machines

Jeff R. Hammond,[*] James Dinan,[*] Pavan Balaji,[*] Ivo Kabadshow,[†] Sreeram Potluri,[‡] Vinod Tipparaju[§]

[*]Argonne National Laboratory. {*jhammond, dinan, balaji*}*@anl.gov*
[†]Forschungszentrum Jülich. *i.kabadshow@fz-juelich.de*
[‡]The Ohio State University. *potluri@cse.ohio-state.edu*
[§]Advanced Micro Devices. *tipparajuv@amd.com*

***Abstract*—Partitioned Global Address Space (PGAS) programming models provide a convenient approach to implementing complex scientific applications by providing access to a large, globally accessible address space. This paper describes the design, implementation and performance of a new one-sided communication library that attempts to meet the needs of PGAS models, particularly Global Arrays, but hopefully also PGAS languages like UPC and CAF. In this work, we describe a new communication runtime for PGAS models such as GA, termed OSPRI (One-Sided PRImitives). OSPRI presents several changes in architecture from conventional one-sided communication systems that make it better suited for emerging leadersip class machines. We describe the implementation of the the IBM Blue Gene/P target for OSPRI and demonstrate significant improvements in latency, bandwidth, and scalability over tuned ARMCI and GA implementations on this system. The performance and scalablity of this library validate the design choices and should provide useful insight for implementers of related communication middleware.**

***Keywords*-Parallel Computing; One-sided Communication; PGAS; Global Arrays; ARMCI**

I. Introduction

PGAS languages and libraries have different runtime system requirements than traditional MPI-based libraries for a number of reasons. The most obvious of these is that the PGAS programming model places a heavy emphasis upon one-sided communication, which was not part of MPI-1 and what was introduced in MPI-2 has been deemed inadequate for at least some PGAS models [5]. To meet the runtime requirements of PGAS models, two portable and open-source libraries have emerged: ARMCI and GASNet. ARMCI was originally part of the Global Arrays library but was refactored as a standalone library and subsequently used to implement a variant of SHMEM [28] and Co-Array Fortran [11], [8]. GASNet was designed specifically for PGAS languages, including Titanium and UPC, but later employed by Chapel and OpenSHMEM. Vendors have their own interfaces for PGAS, which are generally closed-source but occasionally have open API specifications (e.g. Cray DMAPP). While GASNet is generally accepted to be sufficient for PGAS *languages*, it has some shortcomings when compiler support is not present, as is the case for OpenSHMEM and Global Arrays (specifically, lack of strong progress in active-messages, which are the basis for remote atomics and remote accumulate, respectively). In this paper, we describe a new interface for one-sided communication that was designed from the ground up to meet the runtime needs of Global Arrays on leadership computing systems. In the process, many issues were explored that are relevant to other PGAS runtimes, hence we present them here so that others may learn from design strategy and quantification of performance tradeoffs on the Blue Gene/P architecture. In particular, we explored the role of message ordering semantics, thread-safety and interoperability with other runtimes (these are often related), and many aspects of communicating noncontiguous buffers. Special attention is paid to design features that affect performance at extreme scale, which was not just an academic matter since our runtime was used in scientific simulations with nearly 300,000 processes.

In this work, we describe a new one-sided communication runtime for PGAS models termed OSPRI (One-Sided PRImitives). We demonstrate OSPRI's functionality by using it as the communication layer for GA on a leadership class machine and compare its performance with the optimized ARMCI implementation. The implementation OSPRI balances the characteristics of leadership-class machines with the simple and easy-to-use interface provided by the PGAS models. In this paper, we analyze several aspects within the design and implementation of OSPRI on top of Blue Gene/P, including issues related to lightweight operating systems on these machines and their interaction with the hardware and the runtime system, scalability limitations of synchronization mechanisms in PGAS models on large-scale systems, and hardware contention issues that arise due to the shared hardware on these systems. While the work uses GA and BG/P as case studies, we believe that the lessons learned are applicable across PGAS models, and future large-scale computing systems.

It is natural to wonder why we have developed OSPRI in favor of existing models, such as GASNet, ARMCI, MPI, or OpenSHMEM, given that these models are all highly successful in one or more contexts. A summary of the features supported by various runtimes is given in Table I. Further motivation for OSPRI, especially relative to OSPRI, is given in Section VI.

The rest of the paper is organized as follows: Section II provides background information on the PGAS models, GA-

Feature	Progress[0]	Acc.	NB	NC	Atomics
ARMCI	Yes	Yes	No[1]	Yes	Yes
GASNet	No	No[2]	Yes	No[2]	No[2]
SHMEM	Yes	No	Yes	Partial	Yes
OSPRI	Yes	Yes	Yes	Yes	Yes
MPI-2	Yes[3]	Yes	No	Yes	No

Acc. = Accumulate, NB = nonblocking, NC = non-contiguous (strided).

[0] Progress without polling, that is.

[1] This is network dependent and not present in the optimized implementations, e.g. Infiniband.

[2] All of these operations can be implemented as active-messages, but these lack true passive-target progress in GASNet.

[3] Not all implementations support this feature.

TABLE I
FEATURES OF DIFFERENT ONE-SIDED RUNTIME SYSTEMS.

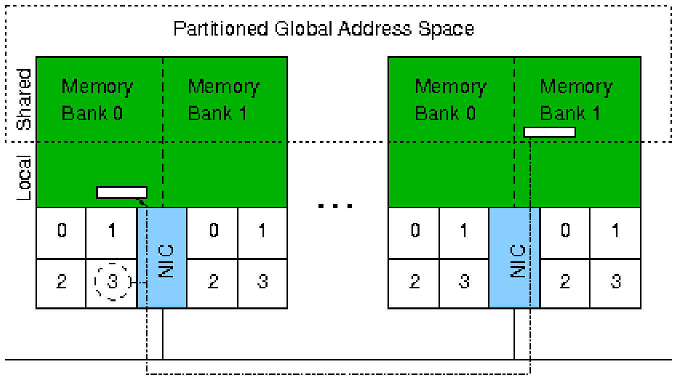


Fig. 1. Example one-sided communication operation. A partitioned global address space can be layered on top of one-sided communication by creating shared regions.

ARMCI in particular. It also presents an overview of the Blue Gene/P system. In Section III, we put forth the limitations of the existing communication run-time architectures like ARMCI and explain the need for OSPRI. Section IV explains how the design of OSPRI differs from that of existing run-times and how these design decisions help address requirements on the leadership-class machines. We describe the device level implementation of OSPRI for Blue Gene/P in section V and evaluate its performance in section VI. Section VIII presents the conclusion and future work.

II. Background

Conventional two-sided communication techniques require one party to perform a `Send()` operation and the other to perform a matching `Recv()` operation. The successful exchange of a two-sided message implies a synchronization between the sender and receiver as the sender must reach the send point in their execution and the receiver must reach the receive point in their program. In addition, the sender and receiver must be expecting to perform the communication and agree on the destination and source for the message, respectively.

For some applications, two-sided messaging can be restrictive due to irregular, data driven communication patterns; data sharing; or computational imbalance between sender and receiver that can lead to high latencies waiting for the message to complete. In order to accommodate applications with these needs, one-sided models have been developed. Models like MPI-2 [18] provide mechanisms for asynchronous one-sided messaging, as shown in Figure 1. One-sided messaging can greatly help applications that exhibit irregular communication and unbalanced computation.

Partitioned Global Address Space (PGAS) models further build on one-sided communication by providing support for a globally accessible shared data space that is spread across all processors. This space grows proportional to the number of processes, enabling applications to process large data sets while providing convenient mechanisms for accessing shared data. Data stored in the global address space is said to have *affinity* to the node in whose memory the data resides and information about affinity and data distribution is made available to the programmer to allow optimizing for local access. Examples of PGAS programming models include UPC [4], Titanium [31], CAF [26], and GA [24]. In addition, the new HPCS languages, Chapel [6] and X10 [7], also provide a PGAS.

A. Global Arrays

In this work, we target the Global Arrays (GA) parallel programming model [24], [29], [21]. GA is a popular PGAS model that provides support for distributed, shared multidimensional arrays and includes a variety of parallel matrix operations including multiplication, diagonalization, and a variety of solvers. GA has been very successful in the computational chemistry domain and is the PGAS model used by the NWChem computational chemistry suite. While quantum chemistry applications are the primary users of GA, GA has been used in other application domains, including groundwater flow modeling, molecular dynamics sampling, among others. More importantly, GA stresses many features of a one-sided runtime, including contiguous and noncontiguous put and get, atomics, and active-messages (remote accumulate for floating-point is usually impossible to do in hardware).

The Aggregate Remote Memory Copy Interface (ARMCI) [20], [23] is the one-sided communication subsystem on which GA is built. ARMCI supports one-sided contiguous, strided, and general non-contiguous get, put, and accumulate operations in addition to a variety of atomic and collective operations. ARMCI is intended to be implemented directly on top of low level networking primitives to take advantage of features like RDMA, however it is also designed to interoperate with MPI and in most situations uses MPI for process management, two-sided messaging, and some collective operations. In addition to GA, ARMCI also serves as the communication layer for Co-Array Fortran [10] and GPSHMEM [27].

IBM systems provide LAPI (Power systems) and DCMF [16] (Blue Gene/P) which possess primitives closely aligned to the needs of GA/ARMCI. ARMCI was implemented for both Blue Gene/L [3] and Blue Gene/P [15] using remote-memory-access (RMA) operations (put/get) and active-messages (accumulate, atomics) provided by these lower-level interfaces.

B. Blue Gene/P

BG/P is the second generation in the IBM BG family. BG/P systems comprise individual racks that can be connected together; each rack contains 1024 four-core nodes, for a total of 4096 cores per rack. Blue Gene systems have a hierarchical structure. Nodes are grouped into midplanes, which contain 512 nodes in an $8 \times 8 \times 8$ structure. Each rack contains two such midplanes. Large Blue Gene systems are constructed in multiple rows of racks. Many details of the BG/P network are given in Ref. [16] so we will not repeat them here.

The compute cores in the nodes do not handle packets on the torus network; the DMA engine offloads most of the network packet injecting and receiving work, enabling better overlap of computation and communication. OSPRI is designed to exploit DMA offload to the maximum extent except when performance is improved substantially by CPU activity. However, the user can disable almost all CPU remote agency for RDMA operations (Put and Get) if this is deemed useful.

The DMA engine on the BG/P maintains a buffer region, known as the DMA FIFO, where it stores data that has been handed over to it by the upper layers but has not yet been reliably transmitted on the network. A process can queue data to be sent on the network by adding it to the DMA FIFO buffer. If this FIFO buffer is full, the process can request the hardware for an interrupt when the DMA engine has transmitted some data, creating more space in the FIFO. On receiving such an interrupt, the process can refill the FIFO with more data. It is up to the user to enable interrupts with an environment variable when asynchronous progress is important and when remote agency is required for progress. Interrupts are not required to make progress on RDMA Put and Get packets.

III. Motivation: The Need for OSPRI

The hardware and software environments of leadership-class systems[1] are undergoing significant changes. Because of this, we must rethink the design of runtime systems that have focused on commodity clusters and retarget this work toward modern, energy efficient and integrated exascale systems. In this section, we describe various characteristics of leadership class machines which make them fundamentally different compared to regular cluster-based systems. For such platforms, communication runtime architectures such as ARMCI present several mismatches.

Massive Parallelism: Systems with a few thousand cores are very common today. Hundreds of thousands of cores are also available on today's largest systems and the next generation of systems are expected to have on the order of several million cores (e.g., the Sequoia system is expected to have 1.6 million cores [1]). At present, the greatest source of increased performance is expected to come from increased levels of parallelism. Thus, forward-looking software must be designed to scale well beyond million-fold parallelism.

Architecturally, a runtime system designed for such platforms cannot have data structures or other bookkeeping that would scale linearly or faster with the number of processes in the system. Thus, hierarchical or multi-level parallelism is fundamental for applications and runtime systems to scale to these systems. While GA provides a scalable communication model for applications using groups and memory subsetting, ARMCI does not fully expose these capabilities to GA. For example, while a process can PUT or GET data from multiple global arrays, ARMCI cannot distinguish the application requiring a completion synchronization (`ARMCI_ALLFENCE`) on one array vs. all the arrays. Similarly, ARMCI requires GA to provide it with a process rank and address for any communication. This notion requires GA to keep track of the shared address region on each node (which scales as $O(N)$ with the number of nodes in the system). Leadership class machines provide capabilities such as symmetric memory allocations where all nodes can allocate a buffer with the same virtual address handle allowing significantly better scalability on these systems ($O(1)$ as compared to $O(N)$), but ARMCI's architecture does not map well to such capabilities.

Scalable Networks: Leadership class systems such as the IBM Blue Gene and Cray XT/XE utilize flat (i.e. scalable) networks which differ from switched fabrics in that they use a 3D torus or similar topology. Although flat networks have cost benefits compared to switched fabrics, they come at the cost of increased network sharing between processing nodes. For example, in a 3D torus, each node has six neighbors that it directly connects to. To reach other nodes, it has to make multiple hops.

Such networks have multiple implications on communication runtime systems. First, these networks provide limited bisection bandwidth. Thus, communication that is not coordinated with other processes in the system would very easily result in network congestion and consequently loss of performance, which makes any communication that can be done in a collectively coordinated manner significantly better than uncoordinated point-to-point communication. Second, because of the high network sharing (and contention) on these architectures, achieving high performance requires communication to take multiple paths simultaneously. This out-of-order communication, however, implies that it gets more cumbersome for the origin process to find out when the data transfer has completed on the remote end.

Tightly Integrated Software Stacks: Instead of general purpose operating system kernels, leadership class systems tend to utilize customized lightweight kernels that avoid unnecessary noise and tend to complement the hardware provided by the system [17]. Specialized operating systems have both advantages and disadvantages relative to Linux, the de facto commodity OS for parallel computing.

ARMCI's communication relies on a data server model, where each physical node utilizes a data server process that

[1] Leadership-class are systems among the largest in size anywhere at a given time; currently, this includes systems with more than 10,000 nodes or 100,000 cores.

exposes the shared memory on a node and allows other processes to read or write data from it. For lightweight kernels such as the IBM compute node kernel on Blue Gene (BG), or the compute node Linux kernel on the Cray XT series, this essentially means that one core in the system has to be dedicated to the data server as these kernels do not allow for automatic multitasking and task scheduling.

These fundamentally different characteristics of leadership class machines as compared to standard cluster machines, warrant a fundamentally different architecture for the communication runtime system, motivating the need for OSPRI.

IV. Design Overview of OSPRI

The design of OSPRI was meant to enable Global Arrays (GA) and the scientific applications built on top of it, especially NWChem, but also applications with different communication needs (e.g. only put/get) and PGAS compilers. The most obvious way to design a runtime for GA is to re-implement ARMCI, since it is the canonical runtime for GA and is an essential component of the GA tools. However, the design of OSPRI varies from ARMCI in a several keys ways: (1) the device specific implementation design allows greater adaptivity to exotic architectures, (2) instead of merely optimizing for IPC (inter-process communication), thread safety is emphasized to effectively support applications usage of hybrid programming models (such as process+threads), (3) relaxed ordering semantics are supported to enable better performance when the application usage permits this, (4) performance oriented settings within the runtime are exposed to enable the applications to adaptively tune the runtime. Note that the application of OSPRI is the library using it (e.g. GA). Additional design differences oriented towards heterogeneous nodes and next generation interconnects will not be discussed in this paper.

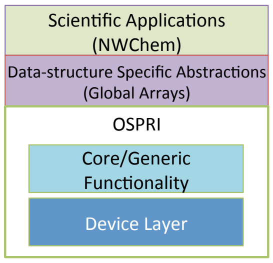

Scientific Applications (NWChem)
Data-structure Specific Abstractions (Global Arrays)
OSPRI
Core/Generic Functionality
Device Layer

Fig. 2. Structure of OSPRI and its Placement in the Software Stack

The design of using a device specific implementation for each architecture is modeled after MPICH2 [19], which has a hierarchical design that allows for maximum code reuse but at the same time permits highly optimized architecture specific functionality when it is justified. In the case of Blue Gene/P, the source branching within MPICH2 is relatively high level due to the close mapping of DCMF to MPI calls and the performance benefit from utilizing this close mapping. The OSPRI design, hence, is hierarchical with the *core* layer/functionality separated from the *device specific* layer/functionality. Of the several design considerations in defining a one-sided communication library, the three most crucial ones are: (1) the data server (or the communication helper thread), (2) message ordering semantics, and, (3) thread safety and hybrid programming support.

A. Aptness of the Data Server

The ARMCI Data Server (DS) serves many functions including [30]: (1) implementing the *accumulate* operation, (2) *pack/unpack* non-contiguous messages, and (3) *read-modify-write (RMW)* and *lock/unlock*. The OSPRI design deliberately excludes DS from the core functionality of the library. There are several reasons for this, these reasons are explained in the context of each of the DS functions listed above.

Accumulate Operation: Element-wise atomic operations are supported by most networks today. However, no architecture supports general purpose floating point accumulate in hardware, hence it is always necessary for either the sender or receiver to perform this computation in the CPU. Because sender-computes requires a round trip transfer, it achieves less than half the bandwidth of a receiver-computes implementation, hence we consider only the latter to be viable for Global Arrays. The sender-computes approach was considered in Ref. [22]. There are two common approaches for invoking remote computation: interrupts and polling. Which of these is better depends on the cost of interrupts versus a dedicated polling thread. Within Linux, interrupts are expensive but oversubscription is well supported. On the other hand, lightweight kernels can provide very efficient interrupts, but, as on Blue Gene's CNK, oversubscription may be impossible and may require a dedicated core.

Pack/Unpack Support: Another important role of the DS is to pack and unpack non-contiguous buffers. This role is less important for networks which allow for high injection rate and support non-contiguous transfer in the low level API. For example, the DMAPP API [2] provided on the Cray XE6 system supports contiguous, strided and indexed Put and Get operations. We expect that substantial hardware support for these operations will render pack/unpack optimizations unnecessary. While Blue Gene/P does not provide any low level hardware or software support for non-contiguous transfer, it is still unnecessary to use a DS because remote unpacking can be implemented within the remote callback in the same manner as floating point remote accumulate. Given the increasing prevalence of both low level support for non-contiguous messages as well as active messages in modern systems, a persistent thread or process such as the DS as a *core* functionality to handle these features is unnecessary.

RMW, Lock/Unlock Operations: Finally, the DS is used for remote atomic operations such as read-modify-write (ARMCI implements only fetch-and-add and swap) and lock/unlock. Once again, increased support for these operations in hardware (e.g. Cray XE6 and IBM PERCS) or through efficient active messages (e.g. Blue Gene/P) makes it unnecessary to implement a generic version through the DS.

Since there are still many scenarios in which one or more communication helper threads (CHTs) improves performance, within OSPRI, the existence and role of CHT(s) is device specific. No core functionality/operations within OSPRI can assume the existence of a CHT and it is desired that device level functionality be CHT agnostic.

B. Message Ordering Semantics

With respect to ordering semantics within OSPRI, three levels of support can be provided that are interesting from a GA perspective. ARMCI provides location-ordered semantics, meaning that not only will sequential overlapping Puts to the same target behave as if they were remote stores, but Accumulate to remote memory followed by a Get from the same memory will provide the desired result. We consider this ordering to be *strict* in the pairwise sense. Although full implementation of ARMCI on top of OSPRI will utilize strict ordering (SO) for all its blocking operation, the GA implementation on top of OSPRI will not. This is because SO is unnecessary for a broad class of usage patterns of GA. In fact, if we go one more level up in the software stack (see Figure 2), NWChem code does almost all of its GA Put, Get and Accumulate calls within well defined epochs bounded by calls to an all encompassing barrier known as `GA_Sync`, which combines the effect of `MPI_Barrier` with remote completion of all outstanding communication operations. Within any given epoch, it can be observed throughout NWChem that only one type of one-sided call is used for a given global array and that most of these calls are blocking. In a blocking Get call, remote completion and local completion are one in the same, hence there is no question of ordering with other Get calls (at least on the same thread). Puts must be ordered within themselves for consistency, whereas Accumulate, which is commutative-associative for all operations defined in the GA API – at least to numerical precision in the case of floating-point – and do not need to be ordered if the algorithm is indifferent to the order of numerical accumulation.

Finally, we consider the unordered (UO) model, within which there is no guarantee of ordering for any operations except via the use of an explicit fence/flush operation. This is the easiest model to provide. Although this may be beneficial on some networks, we find that it is not particularly useful on Blue Gene/P since the only means for enforcing remote completion of Put calls is to send a subsequent zero byte Get (in practice DCMF does this internally) down the same path of the network. Alternatively, one can use dynamic routing with an active message Put (i.e. using `DCMF_Send`) but this then requires remote agency to issue the completion callback, which precludes this mode of operation if both a communication helper thread (CHT) and interrupts are to be considered optional.

We can think of PO as ordering only in the case of WAW (write-after-write) data hazards, while ignoring RAW (read-after-write) and WAR (write-after-read) hazards. However, we must further distinguish between writes that are Puts and writes that are Accumulates. Unlike local stores, remote writes can occur via RDMA or via active-messages wherein the CPU writes to memory. Ordering these two types of operations reduces performance since one must wait for all pending Accumulates to finish before initiating an RDMA Put. It is often cheaper to order RDMA reads and writes than to order the two different types of remote writes. In this respect, ordering semantics for one-sided communication are more complicated than data hazards in shared memory.

C. Thread Safety and Hybrid Programming

Because of the growing popularity of hybrid programming models and improved compiler support for OpenMP, we considered it more pertinent to support a thread safe (in the sense of `MPI_THREAD_MULTIPLE`) API than any specific shared memory optimizations. It is our contention that intranode performance benefits are better realized within a hybrid programming model in conjunction with thread safe OSPRI than through a process model and associated shared memory optimizations. As the number of cores per node increases, it will be increasingly challenging to run in a model that requires one process per core. For a runtime to keep up with all intranode communication is going to become a significant bottleneck due to the memory bandwidth limitations and deep memory hierarchies. Hence the OSPRI design choice of being inherently thread safe is not only more suited for modern programming models but is also carries the ability to scale to dozens of cores per node. Although specific shared memory optimizations are still useful on some architectures today, on Blue Gene/P, it was determined early in the development of OSPRI that the DMA was faster than `memcpy` for intranode transfers larger than 32 KB, so we did not develop any optimizations for interprocess, intranode transfers (e.g. POSIX shared memory).

V. Device Layer Design for Blue Gene/P

The design of OSPRI gives device implementers the freedom to exploit features provided by the underlying system and address any system specific limitations, effectively. In this section we discuss some of the major issues on implementing a one-sided library on the Blue Gene/P system and how these are addressed in OSPRI.

A. True Passive Progress

One-sided communication runtimes are aimed at addressing the requirements of applications/libraries with highly irregular communication patterns. The performance of such applications/libraries largely depends on the runtimes' ability to achieve true passive progress. Blue Gene/P provides a powerful Direct Memory Access (DMA) engine and the Deep Computing Messaging Framework (DCMF) exploits this through truly one-sided communication calls (DCMF_Put and DCMF_Get). However, operations like accumulate, read-modify-write and remote locks are neither supported in the DMA nor provided in the DCMF. These operations have to be executed in callbacks on the remote process. The callbacks are executed either when the remote process calls progress (DCMF_Messager_advance) explicitly or through interrupts. The operating system (CNK) on the Blue Gene/P stores floating point registers as part of the context when an interrupts happens. This leads to flushing a large portion of the L1 Cache and will significantly impact the performance of applications with high data locality. On the other hand, remote process driven progress does not provide the true passive nature desired in a one-sided library.

Blue Gene/P, which has four cores per node, operates in three modes: SMP mode, where only one process runs per node but can launch threads to use other cores, DUAL mode, where two process are launched per node and node-level resources are equally split between them and finally, the VN mode where a process is launched on each core and has a fourth of the node resources. Due to limited memory available on these nodes, many of our target applications, NWChem for example, does not run well in VN mode and its scalability is more limited due to the growth of the local memory footprint with job size. At the same time, these applications are, for the most part, single threaded and are limited in their overall performance when run in SMP or DUAL mode. However, it has be determined through exhaustive performance analysis that NWChem is so communication-intensive at scale that devoting one or more cores to communication (i.e. to run a CHT) reduces the time-to-solution [13], which is the only performance metric relevant that scientists care about. Taking this into consideration, we use a CHT in our design for BG/P. The CHT is a lightweight entity which polls for incoming DCMF active messages by calling DCMF_Messager_advance, making progress on incoming messages.

Thread safety is one of the key design goals of OSPRI and the use of a CHT entails the need for locking inside the library even when the application is single threaded. Though DCMF provides Critical Section functions that can be used to achieve this, we explore the use of BGP Atomics to minimize the locking overhead.

B. Efficient Implementation of Non-Contiguous Transfers

DCMF provides a relatively user friendly API for contiguous transfers that exploit the capabilities of the DMA engine and for general purpose active messages. However, it stops short of implementing non-contiguous functions or read-modify-write like the closely related LAPI API for IBM Power systems. The onus falls on the communication runtime to efficiently implement these operations over the contiguous operations. On the other hand, Global Arrays operates on arrays of two or more dimensions and transfer of sub-matrices (strided) is the most common kind of communication. Hence, effectiveness of non-contiguous transfers is the key to the performance and scaling of GA and its applications. GA depends on the underlying runtime for efficient implementation of these transfers.

A direct way of implementing non-contiguous transfers is to use multiple Put/Get calls, thus achieving true passive progress through use of the DMA. Apart from the overhead of posting multiple messages to the network, such a scheme incurs as significant penalty on systems like Blue Gene/P, where network resources are limited. In our implementation, we reduce the number of messages posted by using packing. The presence of CHT ensures passive unpacking and processing of these messages on the remote side. We maintain buffer pools to avoid the overhead of allocating buffers for each message and these provide us a source side flow control mechanism which is important, especially on a memory limited system like Blue Gene/P. When the buffer pool limit is reached, we stall, and hit on progress until an earlier operation has completed and a buffer has been freed. Such flow control is hard to achieve on the receiver side as we cannot make calls to DCMF advance from inside callbacks, since this would create the possibility of deadlock. The raw network bandwidth on Blue Gene/P is limited when compared to other commodity networks like InfiniBand. Buffering helps show better bandwidth performance to the application as the blocking communication operations can return as soon as the data has been copied over to a buffer.

VI. Experimental Evaluation

The performance of OSPRI has been evaluated on the Blue Gene/P architecture in three ways: (1) OSPRI versus DCMF, to measure software overhead, (2) OSPRI versus ARMCI, for an apples-to-apples evaluation of important one-sided primitives (Put, Get, Acc, Flush) (3) GA benchmarks run using both ARMCI and the OSPRI, which is a drop-in replacement for ARMCI through an identical implementation of all ARMCI functions called by GA. All performance tests were run in SMP mode since there is no additional information to be gleaned from execution in other modes since neither ARMCI nor OSPRI optimizes for this case and we cannot compare thread-safe performance since ARMCI is not thread-safe. Comparison to Berkeley's GASNet communication system was not performed because it was previously demonstrated that it achieves the same low overhead as OSPRI for Put and Get operations on Blue Gene/P [25], while the semantics of GASNet active-messages make them inappropriate to implement ARMCI-style accumulate.

A. Library Overhead in OSPRI

In this section we compare contiguous Put latency in OSPRI with that in DCMF and other one-sided libraries. The amount of overhead each library incurs varies with its various progress and locking modes. We present three modes of operation of OSPRI: (1) helper thread disabled (OSPRI-NoCHT) (2) helper thread enabled and BGP Atomics used for locking to ensure thread-safety (OSPRI-Atomics) and (3) helper thread enabled and DCMF Critical-Section used for thread-safety (OSPRI-CS). Figure 3(L) compares the local completion latency for OSPRI Put with DCMF Put. We see that OSPRI has a 0.4 μsec software overhead relative to DCMF. When the CHT is enabled, lock contention between the main thread and the helper thread incurs some overhead. We see that use of atomics for locking causes lower overhead when compared to using the DCMF Critical Section. However, this only works when OSPRI is the exclusive user of DCMF since only OSPRI can see the Atomic locks used therein, whereas the other clients of DCMF — e.g. MPI or GASNet — will observe DCMF Critical Sections and not make unsafe calls to DCMF simultaneous with OSPRI, provided they are operating in a thread-safe mode. Alternatively, one could explicitly serialize all calls to MPI from within GA using the Atomic locks used by OSPRI, or replace all calls to MPI with calls to DCMF, but this still does not permit GA to be used alongside MPI, which is too important a functional requirement to justify the decreased latency in OSPRI from the use of Atomic locks. All GA results reported herein use DCMF Critical Section rather than Atomic locks to ensure thread-safety.

Figure 3(R) compares the Put latencies in OSPRI with that in ARMCI and MPI. The local and remote completion semantics in OSPRI match that in ARMCI. MPI-2 does not separate local from remote completions. So, we measure the remote completion latency in the passive synchronization mode. We see that OSPRI performs notably better than the both of the other libraries.

B. Global Arrays performance over ARMCI and OSPRI

Remote (inter-process/inter-node) Operations:

In this section, we compare the performance of GA built with OSPRI and ARMCI. For the purpose of obtaining a drop-in replacement for ARMCI which requires only relinking for use, we have created a lightweight ARMCI wrapper so that no modification of GA source was necessary to run tests thereof. In the future, GA will be modified to use OSPRI directly and exploit its unique API features, which we chose not to document in detail here. The performance of remote GA operations is evaluated with one process operating on an array that is distributed across multiple other processes. We have used the standard GA performance benchmarks on four processes where each process operates on a matrix of 1024x1024 doubles. Using more processes for this benchmark is not productive since it is only testing point-to-point communication.

We see that OSPRI reduces the small chunk latency by more than 50% in the case of Put and Accumulate. The Get latency is reduced by 41%. Most of the improvement observed comes from packing, which reduces the number of operations posted to the network. The rest is reduced software overhead because our design is more streamlined by virtue of the device-implementation design. Although OSPRI implements a buffer-pool which allows blocking transfers to return after data is copied into a buffer rather than after local completion, this feature is disabled during benchmarks to show the network performance rather than the speed of local copies. When the buffer-pool is utilized, the CHT handles the communication operations and frees buffers as they are transferred to the network. This is known as handoff mode and it is a runtime configurable setting, as is the size of buffer pools for Get, Put and Accumulate, which are determined separately. The improved performance of OSPRI versus ARMCI for the GA performance benchmark is shown in Figures 4 and 5.

Local (intra-process) Operations:

Global Arrays generates a significant amount of intra-process transfer operations [13] since it is assumed that the underlying one-sided runtime optimizes these appropriately for the system. As noted previously NIC-bypass is not always optimal and on Blue Gene/P, we find that the using DCMF for intra-process is faster than copy for large buffers since it can exploit the DMA and data need not pass through the L1 cache. However, this behavior changes when the DMA is busy with network operations. In that case, memory copy will preform better than the DMA By default, OSPRI uses memory copies (or direct-access in the case of Accumulate) for all intra-process communication. However we have a runtime-configurable option to use DMA operations if it is known to the user that large intra-process transfers will be generated by GA in the absence of significant inter-node communication. The numbers presented in 6(L) compares the performance of local GA operations on a 2D array using ARMCI and OSPRI. We see that for small and medium Put/Get messages the OSPRI memory copy design performs better than ARMCI. As expected, the DMA copies perform better.for very large messages, which is why ARMCI, which does uses DCMF for all local operations, is superior in this regime. Figure 6(R) shows the performance of Local GA Accumulate operations. For Accumulate, there is no advantage of the DMA since the overhead of doing numerical operations within an active-message callback is prohibitive. OSPRI performs significantly better throughout across the message range using direct access.

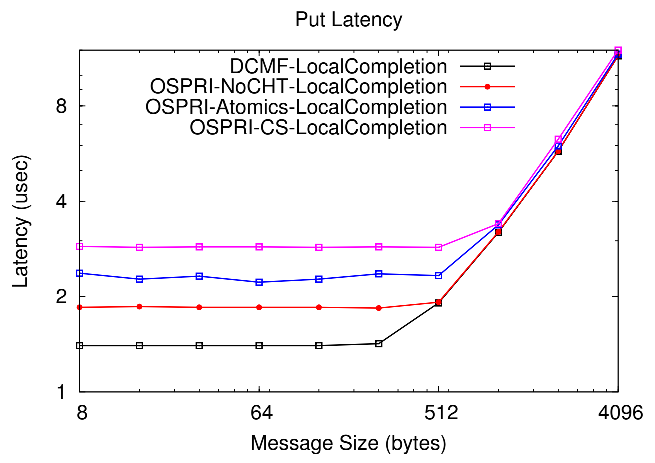
C. Performance effects of OSPRI ordering semantics

The final evaluation of OSPRI is to measure the performance benefit which can be realized with relaxed ordering semantics relative to the strict ordering required for location consistency. As we have not observed a single use case of Global Arrays which requires strict ordering in the one-sided runtime, the contention here is that the improved performance of the partial ordering model realized in the following test is immediately transferable to GA applications such as NWChem.

Figure 7 shows that, for messages as large as 4 KiB, the latency is noticeably greater for Get when using SO versus PO. The overhead comes from the need to flush outstanding Put or Accumulate to the same target, even if the operations are acting on non-overlapping buffers. Of course, one could keep track not only of targets to which Put or Acc messages are outstanding, but also the regions of memory upon which they acted, which, in a communication-intensive application such as NWChem, is either significant memory overhead if stored in a dense fashion or significant processing overhead if stored sparsely.

D. Application Scaling

While it is not the intent of this paper to present the details of our fast multipole method (FMM) implementation using one-sided communication, we briefly summarize the performance results of FMM using OSPRI to demonstrate the viability of this library for real scientific applications, not just microbenchmarks. The full details of our implementation will be describe elsewhere. Figure 8 shows the near-perfect strong scaling of FMM across a 64-fold increase in the number of



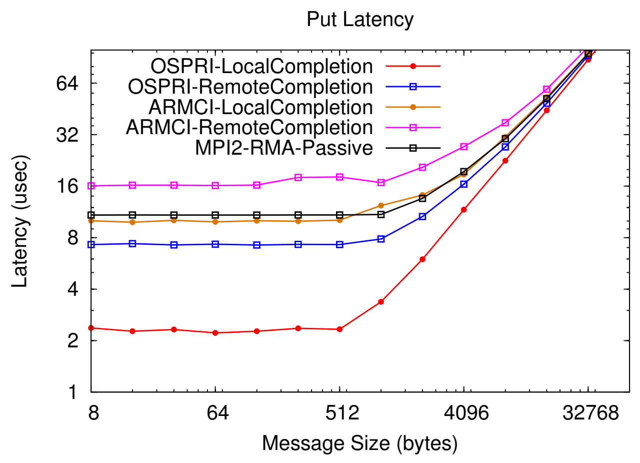


Fig. 3. Put Latency - (L) Comparing DCMF and OSPRI; (R) Comparing OSPRI, ARMCI and MPI

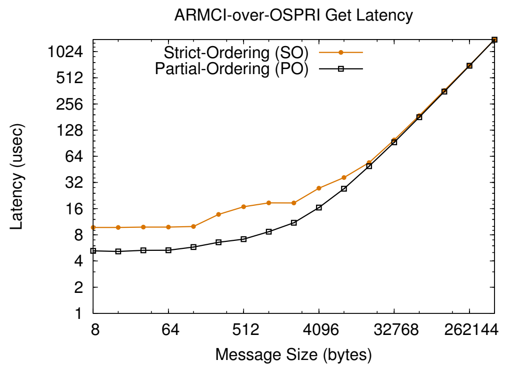


Fig. 7. Latency of ARMCI Get implemented over OSPRI with strict and partial ordering.

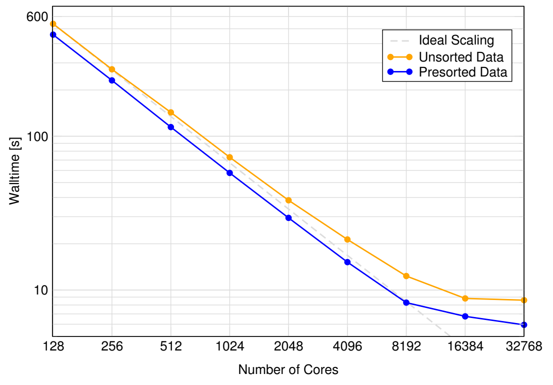


Fig. 8. Strong-scaling of the fast multipole method on Jugene.

processes, to the point at which the performance is saturated. Furthermore, we have performed the largest ever N-body simulations [14], up to 3 trillion particles, using the FMM algorithm on top of OSPRI. These simulations used as many as 294912 cores of Blue Gene/P, which at the time was the largest machine in the world. The timings for ∼1T, ∼2T, and ∼3T particles on 32768 nodes (1 ppn), 73728 nodes (4 ppn) and 73728 nodes (4 ppn) were 2203, 530 and 715 seconds (s), respectively, which is not comparable to any other algorithm or implementation since such simulations are not possible with another method besides FMM nor was it possible to scale the FMM implementation beyond 1024 nodes using MPI-2 RMA or ARMCI. The latter was a strong motivating factor in the development of OSPRI.

While our implementation of OSPRI on Cray Gemini is preliminary, we find that a ∼1B particle FMM solve takes 8.32 s with OSPRI and 22.57 s with ARMCI-MPI, which is demonstrably superior to the native implementation of ARMCI developed by Cray [9].

VII. Related Work

Several one-sided communication substrates have been developed in the context of different higher-level projects. ARMCI [20] and GASNet [4] both support PGAS models; ARMCI supports both GA [24] and Co-Array Fortran [10] while GASNet supports UPC [4], Titanium [31] and Chapel [6]. GASNet is also used as the communication layer for the new Chapel high productivity programming language [6]. The features and shortcomings of these models was discussed previously.

The MPI-2 standard [18] has extended MPI's popular two-sided messaging with one-sided messaging, however restrictions on data access make it unsuitable for supporting a PGAS model. MPI-3 [12] seeks to remedy these limitations with new one-sided primitives.

VIII. Conclusions and Future Work

In this paper we have proposed OSPRI, a new one-sided communication runtime for Global Arrays. The nature and demands of modern leadership class systems was discussed and it was explained how the designs of current runtimes like ARMCI fall short in addressing them. OSPRI has been targeted to overcome these limitations in a modular and scalable fashion. Our work also presents the device-level implementation of OSPRI on Blue Gene/P and shows how the designs provide near-network performance and outperforms ARMCI

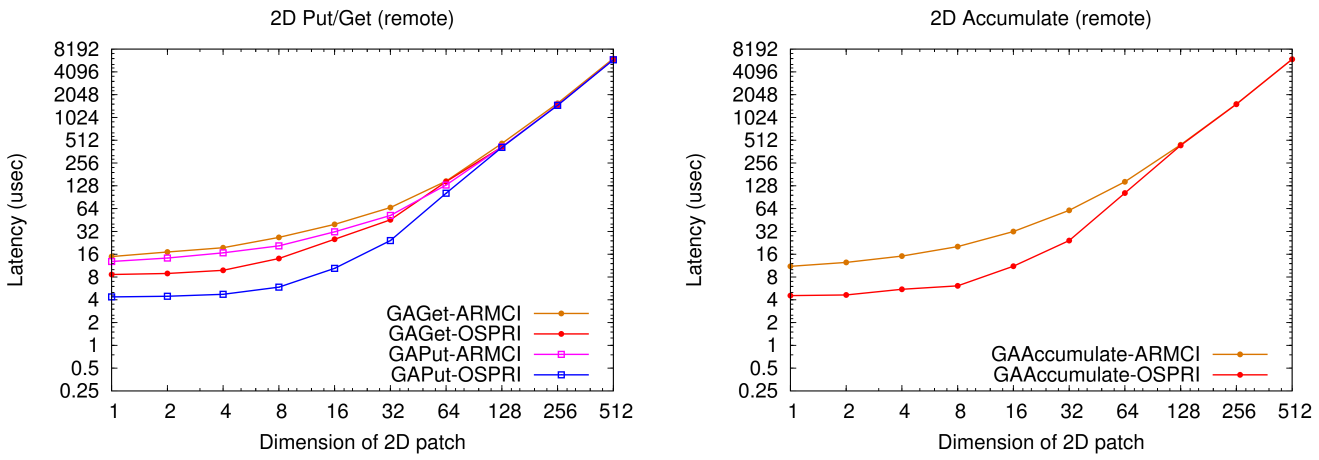


Fig. 4. Latency of GA Remote Operations on a patch of 2D (1024x1024) matrix of doubles: Put, Get and Accumulate.

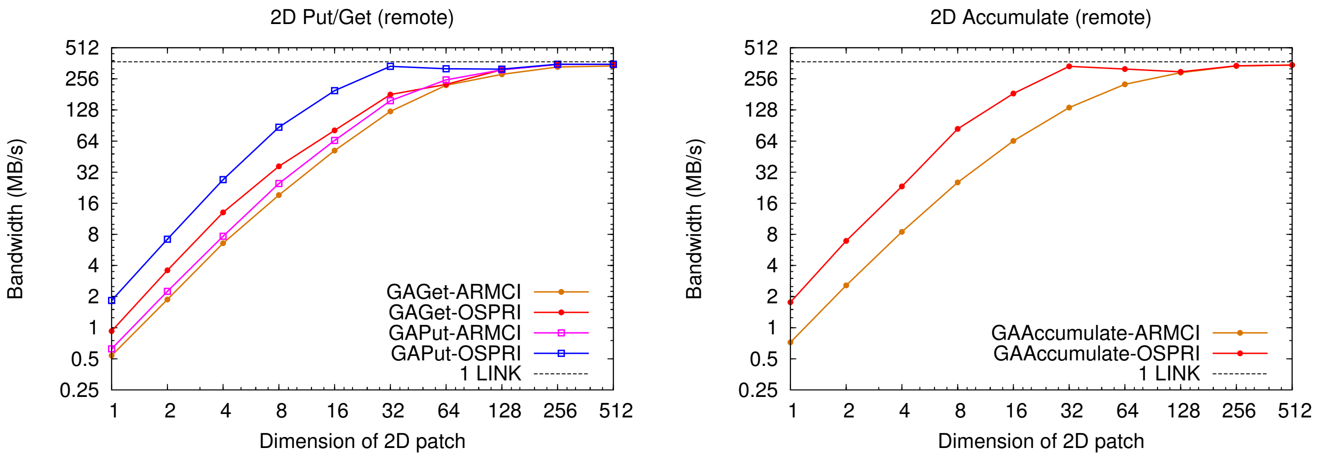


Fig. 5. Bandwidth of GA Remote (Inter-node) Operations on a patch of 2D (1024x1024) matrix of doubles : Put, Get and Accumulate.

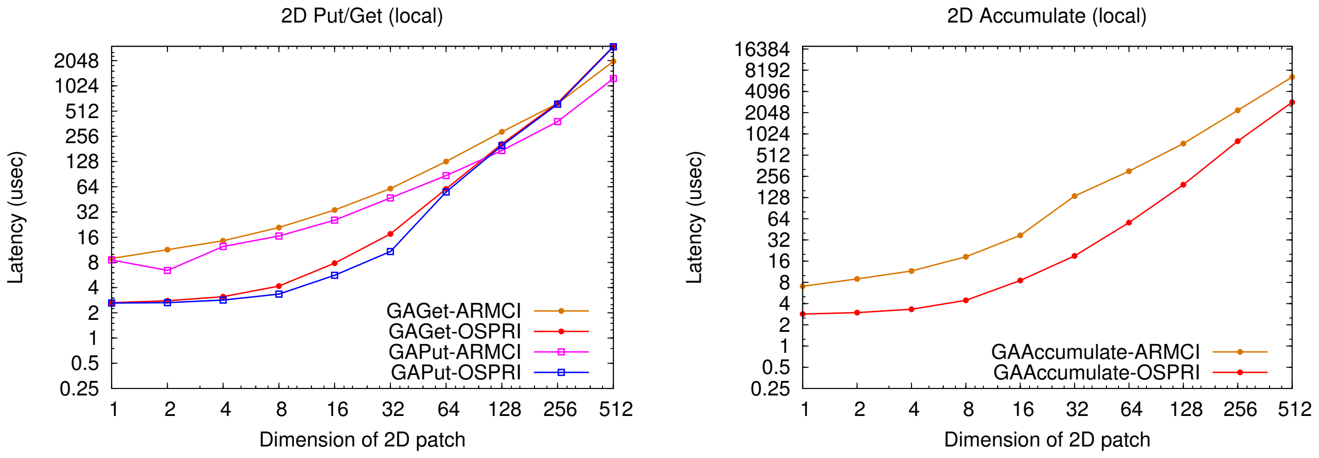


Fig. 6. Latency of GA Local (Same-process) Operations on a patch of 2D (1024x1024) matrix of doubles : (L) Put and Get; (R) Accumulate

as the one-sided runtime system for Global Arrays. This is due to (1) optimizations for noncontiguous operations, (2) better use of CPU and network resources by better optimizing for the architecture, and (3) reduced software overhead due to the device-specific implementation, which shortens the call path relative to a more generic implementation.

IX. Acknowledgments

JRH and PB thank Kazutomo Yoshii for helpful discussions. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] IBM Sequoia. http://en.wikipedia.org/wiki/IBM_Sequoia, July 2010.

[2] Using the GNI and DMAPP APIs. Technical Report S-2446-31, Cray, 2010.

[3] M. Blocksome, C. Archer, T. Inglett, P. McCarthy, M. Mundy, J. Ratterman, A. Sidelnik, B. Smith, G. Almási, J. Castaños, D. Lieber, J. Moreira, S. Krishnamoorthy, V. Tipparaju, and J. Nieplocha. Design and implementation of a one-sided communication interface for the IBM eServer Blue Gene®supercomputer. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 120, New York, NY, USA, 2006. ACM.

[4] Dan Bonachea. GASNet specification, v1.1. Technical Report UCB/CSD-02-1207, U.C. Berkeley, 2002.

[5] Dan Bonachea and Jason Duell. Problems with using mpi 1.1 and 2.0 as compilation targets for parallel language implementations. In *SHPSEC*, pages 91–99, 2003.

[6] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Performance Computing Applications (IJHPCA)*, 21(3):291–312, 2007.

[7] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proc. Conf. on Object Oriented Prog. Systems, Languages, and Applications (OOPSLA '05)*, pages 519–538, 2005.

[8] Cristian Coarfa, Yuri Dotsenko, Jason Eckhardt, and John Mellor-Crummey. Co-array fortran performance and potential: An NPB experimental study. In Lawrence Rauchwerger, editor, *Languages and Compilers for Parallel Computing*, volume 2958 of *Lecture Notes in Computer Science*, pages 177–193. Springer Berlin / Heidelberg, 2004.

[9] James Dinan, Pavan Balaji, Jeff R. Hammond, Sriram Krishnamoorthy, and Vinod Tipparaju. Supporting the Global Arrays PGAS model using MPI one-sided communication. may 2012.

[10] Yuri Dotsenko, Cristian Coarfa, and John Mellor-Crummey. A multi-platform co-array fortran compiler. In *Proc. 13th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, September 2004.

[11] Yuri Dotsenko, Cristian Coarfa, John Mellor-Crummey, and Daniel Chavarría-Miranda. Experiences with co-array fortran on hardware shared memory platforms. In *Languages and Compilers for High Performance Computing*, pages 332–347. IEEE Computer Society, 2005. http://www.springerlink.com/content/67YTT1A3NGQC410L.

[12] MPI-3 Working Group. http://svn.mpi-forum.org/trac/mpi-forum-web/wiki/RmaWikiPage, 2010.

[13] Jeff R. Hammond, Sriram Krishnamoorthy, Sameer Shende, Nichols A. Romero, and Allen D. Malony. Performance characterization of global address space applications: a case study with NWChem. *Concurrency and Computation: Practice and Experience*, 24(2):135–154, 2012.

[14] Ivo Kabadshow, Holger Dachsel, and Jeff Hammond. Poster: Passing the three trillion particle limit with an error-controlled fast multipole method. In *Proceedings of the 2011 companion on High Performance Computing Networking, Storage and Analysis Companion*, SC '11 Companion, pages 73–74, New York, NY, USA, 2011. ACM.

[15] Manojkumar Krishnan, Jarek Nieplocha, Michael Blocksome, and Brian Smith. Evaluation of remote memory access communication on the IBM Blue Gene/P supercomputer. In *ICPPW '08: Proceedings of the 2008 International Conference on Parallel Processing - Workshops*, pages 109–115, Washington, DC, USA, 2008. IEEE Computer Society.

[16] Sameer Kumar, Gabor Dozsa, Gheorghe Almasi, Philip Heidelberger, Dong Chen, Mark E. Giampapa, Michael Blocksome, Ahmad Faraj, Jeff Parker, Joseph Ratterman, Brian Smith, and Charles J. Archer. The deep computing messaging framework: generalized scalable message passing on the Blue Gene/P supercomputer. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 94–103, New York, NY, USA, 2008. ACM.

[17] José Moreira, Michael Brutman, José Castaños, Thomas Engelsiepen, Mark Giampapa, Tom Gooding, Roger Haskin, Todd Inglett, Derek Lieber, Pat McCarthy, Mike Mundy, Jeff Parker, and Brian Wallenfelt. Designing a highly-scalable operating system: the blue gene/l story. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 118, New York, NY, USA, 2006. ACM.

[18] MPI Forum. MPI: A message-passing interface standard. Version 2.2., September 2009.

[19] MPICH2. http://www.mcs.anl.gov/research/projects/mpich2/, October 2010.

[20] Jarek Nieplocha and Bryan Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler runtime systems. In *Parallel and Distributed Processing*, pages 533–546, London, UK, 1999. Springer-Verlag. http://www.springerlink.com/content/P581340602373484.

[21] Jarek Nieplocha, Bruce Palmer, Manojkumar Krishnan, Harold Trease, and Edoardo Aprá. Advances, applications and performance of the global arrays shared memory programming toolkit. *Intern. J. High Perf. Comp. Applications*, 20, 2005.

[22] Jarek Nieplocha, Vinod Tipparaju, and Edoardo Aprá. An evaluation of two implementation strategies for optimizing one-sided atomic reduction. page 7, apr. 2005.

[23] Jarek Nieplocha, Vinod Tipparaju, Manojkumar Krishnan, and Dhabaleswar Panda. High performance remote memory access comunications: The ARMCI approach. *International Journal of High Performance Computing and Applications*, 20(2), 2006.

[24] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: a portable "shared-memory" programming model for distributed memory computers. In *Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 340–349, New York, NY, USA, 1994. ACM.

[25] Rajesh Nishtala, Paul H. Hargrove, Dan O. Bonachea, and Katherine A. Yelick. Scaling communication-intensive applications on Blue Gene/P using one-sided communication and overlap. *Parallel and Distributed Processing Symposium, International*, 0:1–12, 2009.

[26] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[27] K. Parzyszek, J. Nieplocha, and R.A. Kendall. A generalized portable SHMEM library for high performance computing. In *Proc. Intl. Conf. on Parallel and Distributed Computing and Systems (PDCS)*. IASTED, November 2000.

[28] Krzysztof Parzyszek. *Generalized portable shmem library for high performance computing*. PhD thesis, Ames, IA, USA, 2003. Kendall, Ricky A. and Lutz, Robyn R.

[29] Gautam Shah, Jarek Nieplocha, Jamshed Mirza, Chulho Kim, Robert Harrison, Rama K. Govindaraju, Kevin Gildea, Paul Dinicola, and Carl Bender. Performance and experience with LAPI - a new high-performance. In *Communication Library for the IBM RS/6000 SP. In Proceedings of the International Parallel Processing Symposium*, pages 260–266, 1998.

[30] Vinod Tipparaju, Edoardo Aprá, Weikuan Yu, and Jeffrey S. Vetter. Enabling a highly-scalable global address space model for petascale computing. In *CF '10: Proceedings of the 7th ACM international conference on Computing frontiers*, pages 207–216, New York, NY, USA, 2010. ACM.

[31] Katherine A. Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul N. Hilfinger, Susan L. Graham, David Gay, Phillip Colella, and Alexander Aiken. Titanium: A high-performance java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, 1998.