GSHMEM: A Portable Library for Lightweight, Shared-Memory, Parallel Programming

Changil Yoon Vikas Aggarwal Vrishali Hajare Alan D. George Max Billingsley III
ECE Department, University of Florida, Gainesville, FL 32611-6200
{yoon, aggarwal, hajare, george, billingsley}@hcs.ufl.edu

Abstract

As parallel computer systems evolve to address the insatiable need for higher performance in applications from a broad range of science domains, and exhibit ever deeper and broader levels of parallelism, the challenge of programming productivity comes to the forefront. Whereas these systems (and, in some cases, devices) are often constructed as distributed-memory architectures to facilitate easier hardware scalability, some researchers and users believe that programming productivity may be better facilitated with shared-memory programming models. This dilemma may find potential solutions with partitioned global-address-space (PGAS) languages, libraries, and systems. One such PGAS approach is SHMEM, a lightweight, shared-memory programming library originally designed for the distributed-memory Cray T3D machine. With the formation of the OpenSHMEM forum and its upcoming standard, SHMEM is experiencing a resurgence of interest due to its inherent balance in simplicity, programmability, and performance, supported by features such as one-sided communication, an explicit notion of data partitioning and affinity, et al. Unfortunately, SHMEM implementations available to date have largely been proprietary, inconsistent with one another, system-specific, and thus unable to support code uniformity and portability. This paper presents the results of our research investigation, reference design, development, prototyping, and evaluation of a portable OpenSHMEM library (called GatorSHMEM or GSHMEM) that achieves good performance on a potentially wide range of systems, leveraging the GASNet communications middleware from UC Berkeley and LBNL. We evaluate the portability and performance of our approach through microbenchmarking and applications studies on two different systems. Experimental results indicate that our design achieves performance comparable to a proprietary implementation of SHMEM over Quadrics and a popular MPI library (MVAPICH) over InfiniBand.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming-Parallel programming; D.3.2 [*Programming Languages*]: Language Classifications - Concurrent, distributed, and parallel language

General Terms Design, Languages, Performance

PGAS '11 Galveston, Texas.

Keywords Parallel programming; programming model; SHMEM; partitioned global address space; portability

1. Introduction

High-performance computing (HPC) is becoming a critical enabling technology as HPC applications lead to advancements in an ever-broadening range of fields. However, developing such HPC applications is challenging; developers must craft correct and efficient parallel programs that can run on today's complex, highly parallel systems. The programming model used is a crucial aspect of HPC-application development, and the family of partitioned, global-address-space (PGAS) models comprises the current state of the art in balancing performance and programmability. Notable members of the PGAS family include Unified Parallel C (UPC) [1], X10 [2], Chapel [3], Co-Array Fortran (CAF) [4], and Titanium [5].

SHMEM [6] is another member of the PGAS family which takes the form of a library instead of an entirely new language. Originally created for use on Cray systems, SHMEM later became the property of Silicon Graphics Inc. (SGI). The SHMEM library centers on high-bandwidth, low-latency communication routines that take the form of one-sided *put* and *get* operations; these operate on *symmetric* data which resides in logically shared address space. Since it was first created, SHMEM has primarily been available by way of a number of proprietary vendor implementations, notably including versions from Cray [8], SGI, and Quadrics [7]. Unfortunately these implementations have generally differed to varying degrees in the API and semantics they provide, making any level of portability of SHMEM applications—much less the ideal of *performance portability*—very difficult to achieve.

Some previous work [9] has made headway towards a portable SHMEM; GPSHMEM is based on the semantics of Cray SHMEM and uses the ARMCI (Aggregate Remote Memory Copy Interface) library [10] in conjunction with MPI. As the possibility of a more standardized SHMEM library called OpenSHMEM, based on the API owned and controlled by SGI, looms on the horizon, the need for a modern design for a portable SHMEM has again surfaced. A robust design based on the extensive capabilities of the GASNet [11] communication middleware comprises one promising path towards portable performance in the SHMEM community. In this paper, we present the results from our research investigation, reference design, development, prototyping, and evaluation of a portable OpenSHMEM library leveraging the GASNet communications middleware. We evaluate the portability and performance of our approach through microbenchmarking and applications studies on two different systems. Experimental results indicate that our design achieves performance comparable to a proprietary implementation of SHMEM over Quadrics and a popular MPI library (MVAPICH) over InfiniBand. Our reference point for the SHMEM API and semantics is based on our understanding of the V1.0 draft of

the OpenSHMEM specification (shared on March 3, 2011 with OpenSHMEM mailing list).

The remainder of the paper is organized as follows: Section 2 presents background information on SHMEM and GASNet and outlines the software architecture of GSHMEM. Section 3 gives a detailed description of the design of GSHMEM, various routines in our current prototype along with their performance capabilities. In Section 4, we present two application case studies that employ various GSHMEM functions. Finally, Section 5 summarizes the work with conclusions and directions for future work.

2. Background

Traditionally, developers of parallel programs have performed coordination between tasks using either message-passing libraries such as MPI [12] or shared-memory libraries such as OpenMP [13]. Recently, languages and libraries that present a PGAS to the programmer, such as UPC [1] and SHMEM [6], have grown in interest. These languages provide a simple interface for developers of parallel applications through implicit or explicit one-sided data transfer functions, while providing comparable performance to message-passing libraries [15]. In particular, the SHMEM communication library is currently experiencing a growth in interest in the HPC community through the OpenSHMEM initiative. In this section, we provide a brief background of SHMEM and the GASNet communication system which is leveraged by our design of GSHMEM.

2.1 SHMEM

The SHMEM communication library consists of a set of routines that allow exchange of data between cooperating parallel processes (called processing elements or PEs). Programs developed using SHMEM follow the single-program, multiple-data model (SPMD). The SHMEM library includes routines to perform operations such as shared-memory management, data transfers, and synchronization. Table 1 presents some of the most important SHMEM functions, categorized according to the type of operation they perform.

SHMEM supports the PGAS model by defining the notion of symmetric objects, namely those objects (variables or arrays) which have the same size, type and relative address on all PEs. Communication amongst PEs happens only by way of data-transfer operations involving these symmetric objects. Symmetric objects can be separated into two different categories, those that are statically allocated and those that are dynamically allocated. The locations of static variables are assigned at program link time; this property, combined with the SPMD execution model (and the practical reality that the same executable will be run on each PE), ensures that these variables reside at the same virtual address on each PE. SHMEM's dynamically allocated symmetric objects are allocated using a special collective function called *shmalloc()*, which allocates the same amount of memory on each PE (at the same or different memory location).

Table 1. SHMEM functions by category.

Category	*Example Functions*
Environment Inquiry	
Environment Inquiry	_my_pe(), _num_pes()
Data Transfer	
Elemental put/get	shmem_int_p() shmem_int_g()
Bulk put/get	shmem_putmem() shmem_getmem()
Strided put/get	shmem_int_iput() shmem_int_iget()
Synchronization	
Communication sync.	shmem_fence() shmem_quiet()
Wait-on-value	shmem_int_wait() shmem_ int_wait_until()
Barrier	shmem_barrier() shmem_barrier_all()
Group Communication	
Broadcast	shmem_broadcast()
Collective	shmem_collect32()
Reduction	shmem_int_and_to_all()
Atomic Operations	
Atomic swap	shmem_swap()

2.2 GASNet

The Global-Address Space Networking (GASNet) communication system is a language-independent, low-level networking layer developed at U.C. Berkeley and Lawrence Berkeley National Lab (LBNL). GASNet provides network-independent, high-performance communication primitives aimed at supporting SPMD parallel programming models, in particular those in the PGAS family.

At the highest level, GASNet is divided into two layers, the GASNet core API and the GASNet extended API. The Core API is a general interface based on the Active Message (AM) [16] paradigm and Firehose memory registration algorithm [18] implemented directly on top of each network. Active-message-based communication consists of logically matching request and reply messages. Each message is sent with a handler index; when the message is received, the corresponding handler routine is executed. Handlers and their indexes are registered at the beginning of a job execution. A request handler is invoked on a given PE when that PE receives a request message; similarly, a reply handler is invoked upon receipt of a reply message. Request handlers can reply to the requesting node at most one time; reply handlers can neither request nor reply. The GASNet core API also includes functions to: setup and terminate the execution environment; query environmental properties (e.g., processor ID, system size, etc.); send, receive, and execute AMs; and manage locks.

In contrast, the extended API is a network-independent interface that provides medium- and high-level operations such as blocking- and non-blocking remote shared-memory operations (*put* and *get*) and barrier-related operations. Whenever possible, the extended API functions are implemented directly on top of the interconnect API (and thus generally make use of remote direct-memory access or RDMA) to maximize performance. For those networks that do not support RDMA (such as the portable UDP layer), there is a generic implementation that uses only the GASNet core API.

GASNet currently supports execution on a wide range of networks (a.k.a. conduits) such as UDP, MPI, Cray XT Portals, Myrinet, Quadrics, InfiniBand, IBM BlueGene/P DCMF and IBM LAPI [14]. Previous work [15] has demonstrated the performance advantages of GASNet's one-sided communication model compared to MPI's two-sided message passing on BlueGene/P. GASNet continues to be actively developed and used as the network layer for modern programming languages such as Chapel.

3. Design Overview of GSHMEM

Figure 1 shows the software architecture of GSHMEM which employs GASNet's Core API and Extended API. While our interface for GSHMEM uses the OpenSHMEM specification as a reference, we propose few additional routines in this manuscript

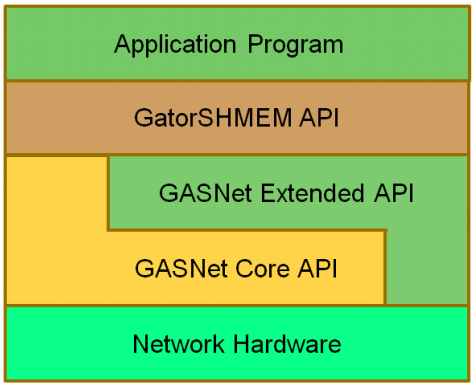

Figure 1. Software architecture of GSHMEM atop GASNet.

as suggestions for inclusion in the future specifications. In the following subsections, we discuss the design of different categories of OpenSHMEM functions in more detail and present benchmarking results to compare the performance of GSHMEM with other libraries. Our current prototype includes all of the functions commonly required by most SHMEM applications. A few functions not included in our prototype are discussed in the final subsection. The results presented in this paper were gathered from experiments on two different systems. The first system (called Quadrics cluster) consists of 16 Linux servers connected via a Quadrics QsNetII interconnect (with a raw link bandwidth of 10Gb/s); each server is comprised of a 2-GHz AMD Opteron 246 processor with 1GB of memory. The second system is an InfiniBand (or IB) cluster and is comprised of 16 servers, each equipped with a quad-core Xeon E5520, 2.26-GHz processor with 6GB of memory. These servers are connected via a DDR-InfiniBand interconnect (offering a raw link bandwidth of 20Gb/s).

3.1 Memory Model

To provide PGAS abstraction in SHMEM, our design relies on the notion and properties of GASNet's shared-memory segments. The GASNet shared segment is an area of memory allocated on each PE (i.e., on each GASNet thread); all remote memory addresses used in GASNet operations must fall within the shared segment. The shared segment is obtained at program startup via the *gasnet_attach()* routine. On parallel multicore systems where each node is comprised of multiple processor cores, each processor core can be configured as an independent PE with a separate shared memory segment. The global address space in such a system is comprised of the shared segments corresponding to various PEs within and across multiple nodes as shown in Figure 2.

Communication among processes on the same node is achieved through GASNet's inter-Process SHared Memory (PSHM) design, which provides three mechanisms namely SystemV shared memory, POSIX shared memory and memory-mapped (using *mmap()*) disk files. GASNET configured with POSIX shared memory over IB restricted the maximum amount of symmetric space available on each PE (multiple of which were mapped on a single node) to a small size. The mechanism using memory-mapped disk files can lead to significant performance degradation on some systems. As a result, we employed the SystemV shared-memory mechanism of PSHM (more details can be found at [19]) to enable support for intra-process communication in GSHMEM. Additionally, GASNet provides various modes for configurations of the shared-memory segment. Our design is based on the use of a configuration in which the segments are aligned at the same virtual address on all PEs (i.e., GASNET_ALIGNED_SEGMENTS). Note that our

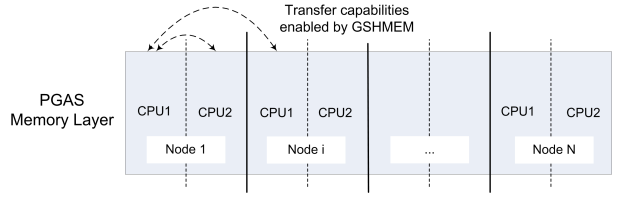

Figure 2. Distribution of global address space in parallel multicore systems.

design does not require the GASNET_SEGMENT_EVERYTHING configuration, which makes the entire memory space of each PE available for remote access instead of a specific shared segment. In addition, PSHM design (on which GSHMEM relies for intra-node communication) provided by GASNET are disabled for the configuration with GASNET_SEGMENT_EVERYTHING. By contrast, with GASNET_SEGMENT_FAST configuration, GASNET supports both shared segment as well as PSHM design. Due to the factors listed in this section, our design for GSHMEM uses GASNET_SEGMENT_FAST.

Dynamic symmetric objects employed by SHMEM applications are allocated directly within the GASNet shared segment. Our design currently leverages a memory management technique and implementation described in [20] for performing *shmalloc()* and *shfree()* operations on the shared segment obtained from GASNet. For static symmetric objects, we employ GASNet active messages in combination with a special portion of the GASNet shared segment reserved for internal buffers in GSHMEM. Further details of this technique will be presented in our discussion of the relevant data-transfer operations.

3.2 Point-to-Point Data-Transfer Operations

The set of point-to-point data-transfer operations in OpenSHMEM consists of elemental, bulk, and strided *put* and *get* operations. The elemental *put* and *get* functions operate on single-element symmetric objects such as short, integer, etc. The bulk *put* and *get* functions transfer a contiguous data block in the form of an array. The strided versions of the data-transfer routines operate on arrays in which the data to be transferred follows a certain pattern based on strides between consecutive elements of the source and target arrays. OpenSHMEM semantics dictate that all *put* routines are non-blocking, i.e. they return as soon as the source data buffer can be reused, while all *get* routines are blocking, i.e. they do not return until the resulting data can be accessed on the calling PE.

In the following subsections, we discuss the transfers involving dynamically allocated symmetric objects and static symmetric objects separately. These cases differ in terms of their design, performance and the GASNet functionality used by each.

3.2.1 Dynamically Allocated Symmetric Objects

Because dynamically allocated symmetric objects always reside within the GASNet shared segment, *put* and *get* routines in GSHMEM that operate on such objects can be based directly on the GASNet's extended API. While there are several choices available for performing *get* and *put* operations in GASNet's extended API (such as blocking, non-blocking, memory-aligned access, etc.), we employed the routines that offered the best performance while conforming to the requirements imposed by OpenSHMEM semantics. For elemental and bulk *get* operations, our design uses *gasnet_get_bulk()* routine. Elemental and bulk *put* operations make use of *gasnet_put_nbi*, which allows us to provide the appropriate (non-blocking) semantics. For strided operations, we employ the indexed routines in GASNet, namely *gasnet_puti_nbi_bulk* and

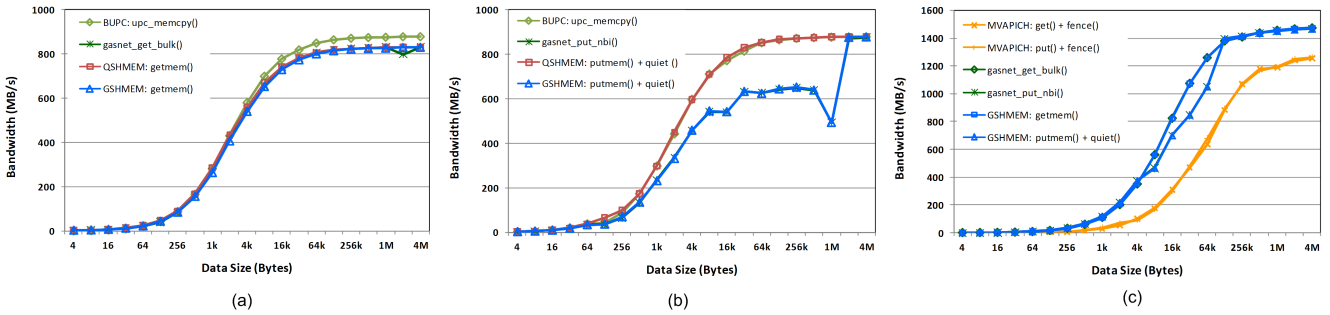

Figure 3. Bandwidth comparison of GSHMEM with other libraries for (a) *get* operation on Quadrics cluster; (b) *put* operation on Quadrics cluster; (c) *get* and *put* operations on IB cluster.

gasnet_geti_bulk. However, the semantics of *gasnet_puti_nbi_bulk* operation are different than that required by OpenSHMEM, as the former does not allow the reuse of source buffer upon return of control to the calling program. As a result, we modified the implementation of *gasnet_puti_nbi_bulk* to provide the desired functionality. An alternate approach for supporting the strided operations can be to employ active messages for accomplishing the transfers. However, such a design based on active messages will require additional memory-copy operations which will lead to performance degradation. More functions to support strided transfers in GASNet in future may help overcome this issue.

To evaluate our design, we compared bandwidth obtained by GSHMEM with several communication libraries on both of the experimental systems (Quadrics and IB clusters). Figure 3a compares the bandwidth obtained by *get* operation in GSHMEM with that of a *get* operations in GASNet (v1.16.1), Quadrics SHMEM (v1.21.2.5) (hereafter referred to as QSHMEM), and Berkeley UPC (v2.12.1)[21] (from UC Berkeley and LBNL, labeled in the figure as BUPC) on the Quadrics cluster. In all of our experiments, we employ the same benchmark code to record the performance of QSHMEM and GSHMEM. In the case of BUPC, the numbers correspond to the bandwidth obtained by a *memcpy* operation between two shared objects. It can be seen that all the communication libraries offered similar bandwidth over a wide range of data sizes.

Figure 3b shows a similar comparison for the *put* operation. In the case of GSHMEM, GASNet, and QSHMEM, the measured bandwidth corresponds to a non-blocking *put* operation followed by a communication synchronization call used to ensure transfer completion. While all communication libraries attained similar bandwidth for data sizes up to approximately 256 bytes, both the GSHMEM *put* operation and GASNet's *put* have lower bandwidth for data sizes between 256 bytes and 2MB. To find possible reasons for this observed discrepancy, we explored the design and implementation of the corresponding GASNet *put* operation on the Quadrics cluster (i.e., for the Quadrics ELAN conduit). We found that for data sizes up to 64 bytes, the *put* operation is implemented by directly calling an ELAN *put* operation. However, for data sizes between 64 bytes and 1MB, the GASNet *put* implementation performs an explicit copy from the source buffer into an ELAN bounce buffer (from which the network will presumably later perform an RDMA-based transfer). For data sizes over 1MB, the GASNet *put* operation directly employs active messages. Based on these observations, we surmise that the copy operation being used for the intermediate data sizes leads to the observed performance discrepancy. Future work will explore possible avenues for mitigating this issue for our design of the GSHMEM *put* operation.

Figure 3c compares the performance of *get* and *put* operations in GSHMEM with corresponding operations in GASNet on the IB cluster. Due to the lack of a SHMEM implementation over

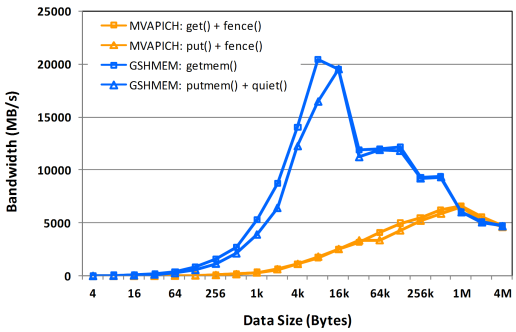

Figure 4. Bandwidth comparison of intra-node *put* and *get* transfers on IB cluster with similar transfers using MVAPICH.

InfiniBand, we also compare GSHMEM with MVAPICH2 (v1.4.1) (a popular implementation of MPI over InfiniBand) [17]. Note that we employ the one-sided MPI operations such as *MPI_Get* and *MPI_Put* in combination with *MPI_Win_fence* in our experiments on IB cluster, in order to make a fair comparison with other one-sided communication libraries. As with the Quadrics cluster, the IB cluster shows little observable difference in performance between GSHMEM and GASNet, suggesting that GSHMEM has small overhead regardless of the execution platform. Additionally, on the IB cluster, GSHMEM *put* and *get* operations outperform MVAPICH across all measured data sizes above 256 Bytes (all libraries are comparable below this range). Figure 4 compares the bandwidth obtained by GSHMEM and MVAPICH for data transfers between two processor cores within a single node. The figure indicates that GSHMEM is capable of offering much superior performance for intra-node transfers by employing the GASNet's PSHM design.

3.2.2 Statically Allocated Symmetric Objects

Since static symmetric objects are not allocated within the GASNet shared segment, the extended API of GASNet cannot be directly employed for these objects. Instead, GSHMEM uses active messages to perform remote transfers to static symmetric objects. We illustrate our design by considering the example of bulk transfer routines *shmem_getmem()* and *shmem_putmem()*, for which we make use of GASNet long active messages. Since long active messages perform data transfers only within GASNet's shared segment, our design reserves a portion of the GASNet shared segment as internal buffers during initialization. The size of this buffer is user-configurable, with a default size of 64MB in our current implementation.

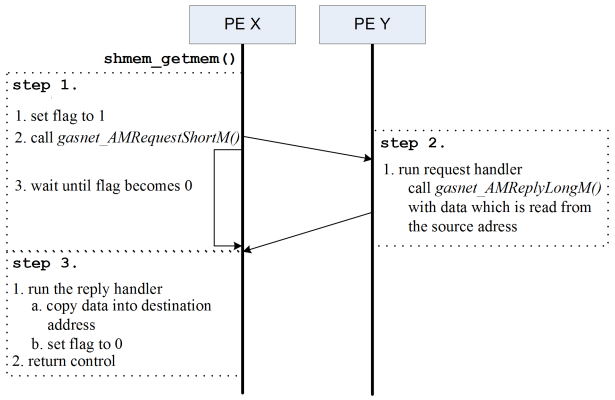

Figure 5. Sequence of steps for *shmem_getmem()* routine involving statically allocated data.

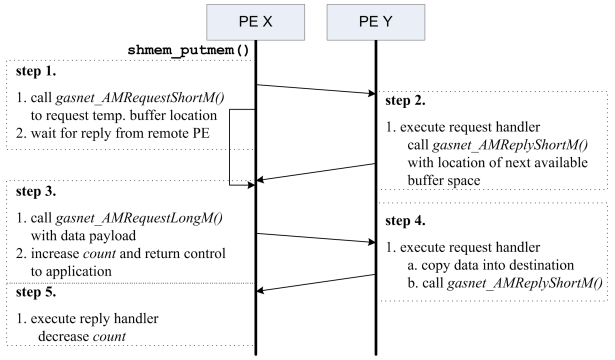

Figure 6. Sequence of steps for *shmem_putmem()* routine involving statically allocated data.

Figure 5 illustrates the message flow in our design when a *get* operation is invoked on PE X to transfer a static symmetric object residing on a remote PE Y. The following is a simple description of this message flow:

1. PE X first initializes the flag to 1, and then invokes *gasnet_AMRequestShortM()* to send the address of source buffer to PE Y. PE X then waits until the value of the flag becomes 0 before returning control to the application program.
2. After receiving the request, PE Y executes the associated request handler, which in turn ultimately calls *gasnet_AMReplyLongM()* to send requested the data from the source address on PE Y.
3. Upon receiving the reply message, PE X executes its reply handler, which copies the data into the destination address and then sets the value of the flag to 0. At this point, *shmem_getmem* returns control to the application program on PE X.

Unlike the *get* operation, the *put* operation is non-blocking and returns before the data transfer is completed. To accomplish this capability, we employ an internal buffer on the destination PE to store the data temporarily. Figure 6 shows the message flow for a *put* function called on PE X where the destination is a statically allocated symmetric object residing on remote PE Y. This message flow is as follows:

1. PE X sends a *gasnet_AMRequestShortM()* to request the location of the available buffer on PE Y and waits for a response.

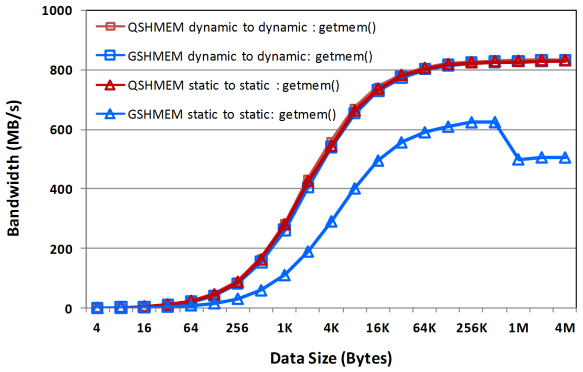

Figure 7. Bandwidth comparison between QSHMEM and GSHMEM for data transfers using statically and dynamically allocated symmetric objects on Quadrics cluster.

2. The request handler on PE Y, responds with the address of the next available buffer space through a *gasnet_AMReplyShortM()*.
3. PE X invokes *gasnet_AMRequestLongM()* to transfer the data (to the temporary buffer on PE Y) and send the destination address to PE Y. Additionally, PE X increments its *count* of outstanding *put* operations and returns control to the caller immediately. *count* is used by synchronization routines in our design to check for any incomplete put operations.
4. Upon receiving the message containing the data, PE Y calls the associated request handler which copies the data into the destination address. As the last step within the request handler, PE Y calls *gasnet_AMReplyShortM()* to indicate that the completion of transfer.
5. On receiving the reply, PE X executes the corresponding reply handler, which decrements the count of outstanding *put* operations.

Figure 7 shows the bandwidth of QSHMEM and GSHMEM when transferring data between two PEs using dynamically allocated symmetric objects (the *dynamic-to-dynamic* case) and using statically allocated symmetric objects (the *static-to-static* case). While in the dynamic-to-dynamic case GSHMEM and QSHMEM offer comparable performance, for the static-to-static case GSHMEM offers lower bandwidth compared to QSHMEM. This discrepancy can be explained based on our current GSHMEM design (and prototype), which uses an explicit *memcpy* call to copy data from the remote shared segment to the remote static-memory location (i.e., step 2 of *shmem_getmem()* discussed in Section 3.2.2). While this copy operation is necessary in the current design, future work will explore possibilities for solving or at least mitigating this issue.

3.3 Collective Communication

The OpenSHMEM library provides a set of collective routines including broadcast, collect and reduce operations. These operations can be either performed on all the PEs in an application or amongst a specific subset of PEs called an active set defined by a triplet: starting PE, stride and size of the active set. Each of these collective operations can be internally mapped to one or more collective operations recently supported in GASNet's extended API.

For OpenSHMEM's *broadcast* routine, our design uses *gasnet_coll_broadcast*. Similarly, *fcollect* function is built on top of GASNet's *gasnet_coll_gather_all* routine. While OpenSHMEM semantics require that the result of a reduce operation be available

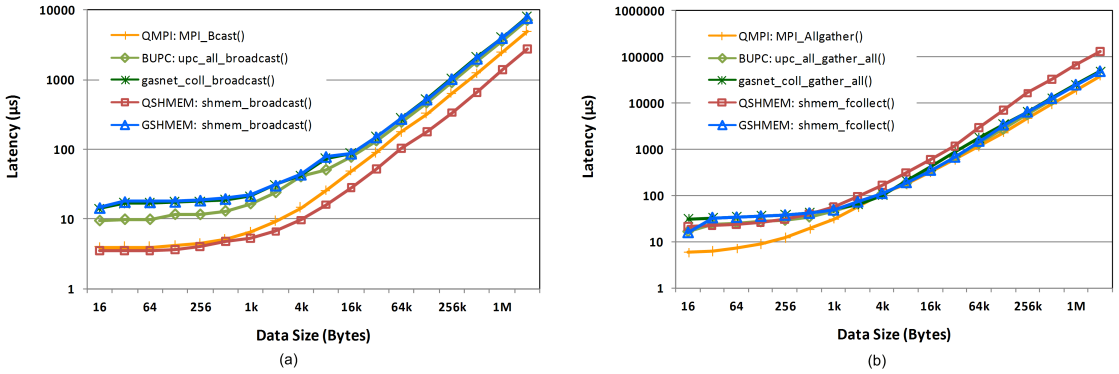

Figure 8. Latency comparison of GSHMEM with various communication libraries on Quadrics cluster for (a) *broadcast* operation; (b) *fcollect* operation. Experiments involved eight nodes exchanging data.

on all the participating PEs, the reduction operation in GASNet (*gasnet_coll_reduce*) updates the result on the root PE only. As a result, our design performs a subsequent broadcast operation to broadcast the reduced data to all the PEs at completion of *gasnet_coll_reduce*.

For defining an active set for any collective operation, we employ GASNet's notion of teams. A team in GASNet is defined by a team handle, and is created using *gasnete_coll_team_create*. Once a team is created, the corresponding team handle can be employed by GASNet collective routines to perform communication between the PEs participating in that team. To minimize the overhead of repeatedly creating teams, our design caches the most recently-used team handle. Our performance benchmarking indicated that such caching can significantly reduce latency, and can be beneficial when collective operations are called repeatedly with same active set (a common occurrence in SHMEM applications).

To further optimize the performance of collective operations in OpenSHMEM, we (optionally) employ GASNet's auto-tuning infrastructure for collectives. When enabled, this feature initiates a search for an optimal algorithm for executing a given collective operation (defined by the routine and its associated parameters). Environment variables *GASNET_COLL_ENABLE_SEARCH* and *GASNET_COLL_TUNING_FILE* are used to control the behavior of auto-tuning framework in GASNet. We observed substantial improvement in the performance of collective operations by employing auto-tuning during our experiments, especially for large data transfers.

Figure 8 compares the latencies of *broadcast* and *fcollect* operations in GSHMEM with equivalent operations in QSHMEM, GASNet, and Berkeley UPC on the Quadrics cluster. For collective operations, we also compare the performance of GSHMEM with that of corresponding operations in an implementation of MPI from Quadrics (v1.2.4) running on top of the Quadrics network API, labeled QMPI in the figure. The results presented in the figure for GSHMEM, GASNet and BUPC correspond to the performance obtained by enabling the auto-tuning infrastructure of GASNet. For *broadcast* operation (Figure 8a), GSHMEM continues to offer very little overhead on top of GASNet. However, the three communication libraries (i.e. GSHMEM, BUPC, and GASNet) that employ GASNet functionality offer lower performance compared to vendor-specific libraries from Quadrics (i.e. QSHMEM and QMPI). After further investigation, we determined that the broadcast operation in QSHMEM uses highly-optimized hardware broadcast primitives when the operation involves a contiguous set of PEs. For *fcollect* operation (Figure 8b), the performance of GSHMEM compares favorably to that of QSHMEM for message sizes greater than 2KB, while still incurring minimal overhead on top of GASNet.

We also compared the performance of GSHMEM and QSHMEM for collective operations performed on a subset of PEs in a SHMEM application. For such team-based operations, both *broadcast* and *fcollect* in GSHMEM were able to outperform their counterparts in QSHMEM for large message sizes as shown in Figure 9a. For smaller messages, the overhead incurred by GASNet in handling team-based collectives and GSHMEM for creation of a new team through GASNet led to lower performance compared to QSHMEM.

Although the draft OpenSHMEM specification does not provide scatter and gather functions currently, we have included these functions in our interface of GSHMEM. We believe these functions can potentially improve performance and productivity (in certain cases) for some applications. Consider the case of *fcollect* operation in OpenSHMEM, which requires that the resultant (collected) data be available on all the PEs in an active set. Several applications only require the resulting data to be available on one of the PEs, and a *shmem_gather* (if supported) will provide the desired functionality. Employing a *fcollect* operation in such a case can lead to unnecessary data transfers, yielding sub-optimal performance. To provide application developers with more flexibility, we provide *scatter* and *gather* functionality in GSHMEM. These functions are directly built on top of their counterparts available in GASNet. Figure 9b compares the latency of scatter operation in GSHMEM with corresponding operations in GASNet, BUPC and QMPI. All the communication libraries offer similar performance over a wide range of data sizes.

3.4 Synchronization

The synchronization routines provided by OpenSHMEM can be grouped into three categories: communication synchronization, barrier synchronization, and point-to-point synchronization (through wait-on-value-change operations). In the following subsections we describe our design for each of these categories of synchronization routines.

3.4.1 Communication Synchronization

Since *put* operations in OpenSHMEM are non-blocking, programmers need a way to ensure completion of outstanding *put* operations and to establish ordering of multiple *put*s. For this purpose, OpenSHMEM provides the *shmem_quiet()* and *shmem_fence()* routines. *shmem_quiet()* blocks the program execution for the calling PE until all outstanding *put* operations (to any PE) are completed. *shmem_fence()* operation is employed to guarantee the ordering

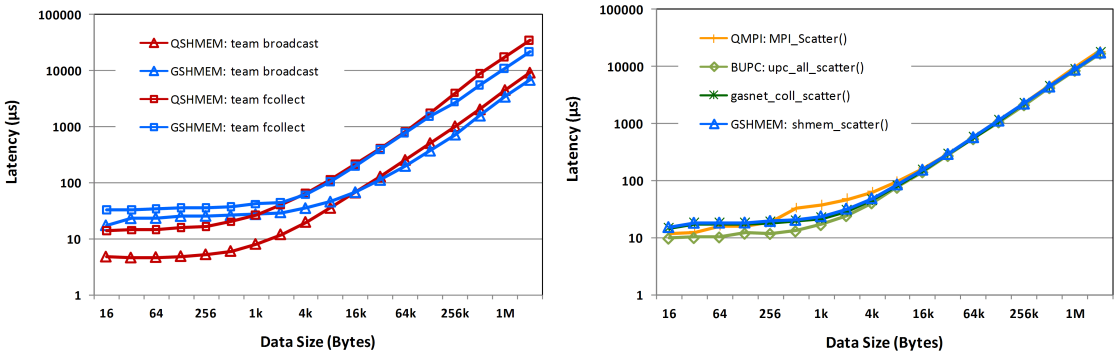

Figure 9. Latency comparison of GSHMEM with various communication libraries on Quadrics cluster for (a) team-based collective operations; (b) *scatter* operation. Experiments involved eight nodes exchanging data. For team-based operations, four out of eight nodes were chosen using startPE=0, stride=1, size=4.

of *put* operations to a particular PE. In this section, we address the design of *shmem_quiet()* in GSHMEM. Our current design of *shmem_fence()* simply provides the same (stronger) semantics of *shmem_quiet()*. A more refined design for the fence operation is left for future work (though we have a sketch of this design already completed).

As described in Section 3.2.1, our design for *put* operations involving dynamically allocated symmetric objects is based on a GASNet extended API routine which uses an implicit handler. Thus, our corresponding design for *shmem_quiet()* uses *gasnet_wait_syncnbi_puts()*, which waits until all the GASNet *put* operations issued by the calling PE are completed. To ensure that *shmem_quiet()* also waits for completion of *put* operations to statically allocated symmetric objects, we make use of the counter mentioned in Section 3.2.2. Our *shmem_quiet()* simply ensures that this counter is zero, indicating that no *put* operations to static objects on remote PEs are outstanding.

3.4.2 Barrier Synchronization

The OpenSHMEM library provides two types of barrier synchronization routines, *shmem_barrier_all()* and *shmem_barrier()*. While the first of these functions, *shmem_barrier_all()*, performs a barrier operation on all of the PEs in the application, *shmem_barrier()* routine performs a barrier operation for a subset of PEs in an active set (specified by a triplet: start PE, stride, and the size of the active set).

Our design of *shmem_barrier_all()* makes use of the split-phase barrier routines provided by GASNet, namely *gasnet_barrier_notify()* and *gasnet_barrier_wait()*. By invoking the *wait* operation immediately after the *notify* on each PE, we obtain the collective, single-phase barrier behavior needed for *shmem_barrier_all()*. For performing a barrier operation on a subset of the PEs using *shmem_barrier()*, we define a GASNet team as described in Section 3.3, and then employ team-based GASNet routines for notify (*gasnete_coll_teambarrier_notify*) and wait (*gasnete_coll_teambarrier_wait*).

3.4.3 Point-to-Point Synchronization

The SHMEM library provides two functions for synchronization between two PEs, *shmem_wait* and *shmem_wait_until*. The *shmem_wait* routine blocks until the specified variable is changed to a value other than a specified value by a remote PE. The *wait_until* functions does not return until the given comparison statement (comprised of a symmetric object, a value, and a comparison operator) is satisfied by a change from a remote PE.

Our design of the *shmem_wait* and *shmem_wait_until* routines for statically allocated symmetric objects makes use of the GASNet function called *GASNET_BLOCKUNTIL()*, which waits (busy polling for active messages) until a given condition becomes true by way of some incoming active message. Since transfers to dynamically allocated symmetric objects do not employ active messages in our design, we use explicit polling for *wait* operations involving dynamic objects. Such polling is achieved in GSHMEM by invoking a local GASNet *get* operation to ensure that we see any changes to the applicable symmetric object.

3.5 Setup and Environment-Query Operations

The GSHMEM library provides a function for initialization (*start_pes()*) of execution environment. This function internally performs all the necessary tasks to initialize the operation of GASNet. While not present in the OpenSHMEM specification, we also include a corresponding routine (*shmem_finalize()*) for termination. We believe that such a routine can be beneficial for ensuring a clean exit for SHMEM applications, performing housekeeping activities (such as the ones require by GASNet's autotuning feature), and a capability important to support performance-analysis tools for the OpenSHMEM library.

OpenSHMEM additionally provides environment query functions, including *_my_pe()*, which simply returns the PE identifier for the calling PE, and *_num_pes()*, which returns the total number of PEs. Our design for these routines simply makes use of corresponding GASNet functionality, namely *gasnet_mynode()* and *gasnet_nodes()*.

3.6 Open Issues

The OpenSHMEM specification includes routines to perform atomic read-and-update operations (e.g., fetch-and-increment) on symmetric data objects. We have a preliminary design for such operations which exploits the atomicity of active-message handlers. Due to the lack of clear semantics and disparity between potential choices for such operations, GSHMEM does not include a detailed design for such operations currently. Additionally, the OpenSHMEM library provides a set of miscellaneous functions for purposes such as cache management and address manipulation, which we plan to investigate in future.

4. Application Case Studies

In this section, we demonstrate the portability and performance of GSHMEM by way of two application case studies which we

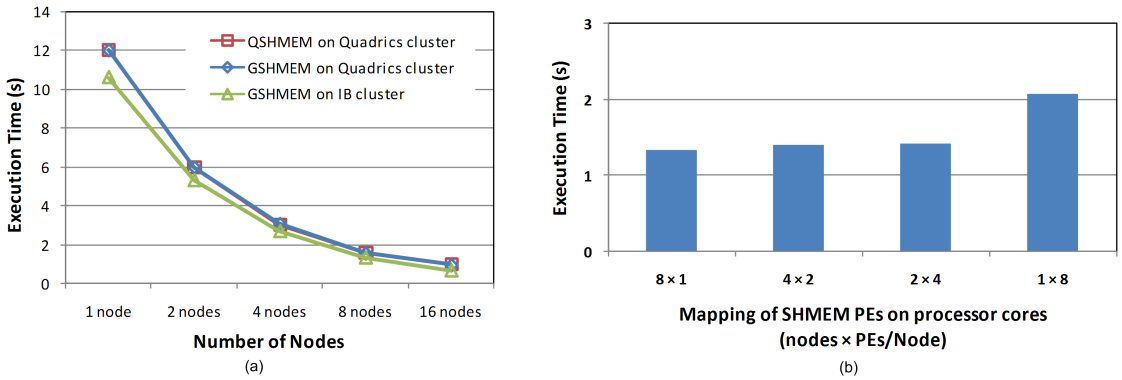

Figure 10. (a) Execution time of CBIR application using QSHMEM and GSHMEM on Quadrics and IB clusters;(b) Execution time of CBIR applications with 8 PEs using different processor assignments. For all experiments, our search database consisted of 22,000 images, each of size 128×128 pixels of 8 bits.

developed to conduct our experiments. These studies illustrate various functionalities of GSHMEM such as the use of GSHMEM routines for transfers between PEs mapped on processors on the same node or different nodes, as well as comparison of different routines in GSHMEM when performing certain types of transfers. Additionally, we compare the performance of an application developed using GSHMEM with that of QSHMEM and demonstrate the ability of GSHMEM to provide portable performance by running the same application on two distinct systems (Quadrics cluster and IB cluster). Due to lack of available SHMEM applications, our experiments have been limited to two case studies. We plan to expand our set of application case studies in future.

4.1 Content-based Image Retrieval (CBIR)

The first application we considered is a SHMEM implementation of Content-Based Image Retrieval (CBIR), which refers to the efficient search of multimedia databases based on the semantics of the data. The images in the database are characterized by a feature vector later used for retrieval of relevant images. For this case study, we employed a color-feature-extraction CBIR program developed based on the auto-correlogram discussed in [22].

The execution time of the application on the Quadrics cluster using both GSHMEM and QSHMEM is shown in Figure 10a. The figure also shows the execution time of the application using GSHMEM on the IB cluster. The graph indicates that GSHMEM is capable of providing application performance similar to that offered by a vendor-provided, system-specific SHMEM on a cluster with a proprietary interconnect (QSHMEM over Quadrics in our experiments). Additionally, GSHMEM allows applications to be immediately ported to another cluster with an unrelated interconnect technology as demonstrated by the ease with which we were able to run the same CBIR application on the IB cluster.

To analyze the ability of GSHMEM to transfer data between PEs mapped on processors within a node, we conducted experiments using the CBIR application with eight PEs on the IB cluster which consists of quad-core processors on each node. Figure 10b shows the execution time of the application under different mapping scenarios. The labels for different bars in the graph indicate the number of nodes employed in an experiment $\times$ the number of PEs mapped on each node. GSHMEM allows the application developers to express communication between different processor cores within a node and across multiple nodes using a uniform interface. The first three cases (8×1, 4×2 and 2×4) offer similar performance. Due to the small amount of communication involved in the CBIR application (compared to computation time), higher bandwidth of intra-node transfers does not lead to any performance improvement by mapping more PEs on the same node. For the case of 1×8 mapping (last bar in the graph), two PEs time-share the resources of each processor core (through Intel's Hyper-Threading technology), which results in a higher computation time for the application. This study illustrates the ease with which GSHMEM applications can be executed on various processors cores in a cluster of multicore processors.

4.2 Two-dimensional FFT

As our next case study, a parallel, two-dimensional FFT was chosen because of a more complex communication pattern which can be provided in multiple ways using different SHMEM routines. A 2-D FFT operation on an image is performed by decomposing it into a series of 1-D FFT over the rows of the image, followed by a series of Fourier transforms over the columns. A typical parallel implementation of 2-D FFT distributes rows of the input image across the computational nodes which perform a 1-D FFT over their assigned subset of rows. A corner-turn (distributed transpose), which involves all-to-all communication between the processing nodes, is required to re-distribute the data across all the nodes. The nodes then compute 1-D FFT over the columns of the image. Another corner turn is required to re-organize the data and recover the transformed output image. In our experiments, we employed GSHMEM routines to perform the corner-turn in three different ways: (a) using a *shmem_fcollect* operation, (b) using a *shmem_gather* operation, and (c) using a *shmem_get* operation multiple times to receive data from every PE.

Figure 11 shows the execution time of 2-D FFT for three different implementations using GSHMEM (Figure 11a) and QSHMEM (Figure 11b) on the Quadrics cluster. The results indicate that GSHMEM offers application performance comparable to that of QSHMEM for the approach using *get* operation and much better performance than QSHMEM for the approach using *fcollect* operation (this is because the performance of *fcollect* operation in GSHMEM was shown to be better than QSHMEM in Figure 8b). Additionally, this study illustrates the benefits of including a *shmem_gather* routine in the OpenSHMEM specification. For this particular application which requires resultant data to be copied on only one PE, a *fcollect* operation leads to unnecessary data transfers compared to a *gather* operation, it exhibits poor performance and scaling behavior when compared to the latter. Interestingly, the implementation using a *get* operation offered the best performance in our experiments. By virtue of being a non-collective operation,

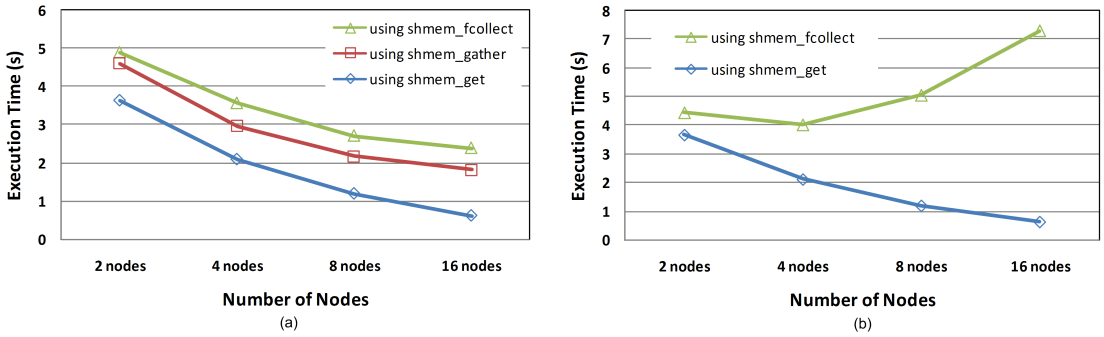

Figure 11. Execution time of two-dimensional FFT implemented on Quadrics cluster (a) using different GSHMEM functions; (b) using different QSHMEM functions.

the *get* operation allows each PE to essentially perform a gather operation in parallel with a similar operation on other PEs.

5. Conclusions and Future Work

As parallel systems evolve towards large clusters of multicore and manycore processors, communication libraries that are capable of offering high-performance, portability and programming simplicity will become essential. To address this need, we have presented our design for a portable, high-performance SHMEM library atop the GASNet system. The substantial portability which can be achieved with the GASNet layer, combined with the available performance on the wide range of networks for which optimized GASNet support is available, make GASNet particularly amenable for a modern, portable SHMEM design and implementation.

Our current GSHMEM reference design and prototype provides the core set of OpenSHMEM API functionality. We demonstrated the portability and performance of GSHMEM through microbenchmarks and two case studies on two distinct systems with different interconnect technologies. We have also shown the performance of GSHMEM to be comparable with the performance of the vendor Quadrics SHMEM implementation in most cases.

Future work on GSHMEM will focus on completing all remaining design considerations to support the complete OpenSHMEM API as mentioned in Section 3.6. We also plan to further explore and optimize the performance of our design, as well as improve the usability and robustness of our prototype to yield a more complete software system. Finally, we plan to provide support for performance analysis of GSHMEM applications, in particular using our own Parallel Performance Wizard (PPW) tool [23].

Acknowledgments

This work was supported in part by the U.S. Department of Defense and Lawrence Berkeley National Lab.

References

[1] Carlson, W.W., Draper. J.M., Culler, D.E., Yelick, K., Brooks, E. and Warren, K. *Introduction to UPC and language specification.*University of California-Berkeley. Technical Report: CCS-TR-99-157, 1999.

[2] Murthy, P. *Parallel computing with x10.*In Proceedings of the 1st international Workshop on Multicore Software Engineering (Leipzig, Germany, May 11 - 11, 2008).IWMSE '08. ACM, New York, NY, 5-6.

[3] *Chapel parallel programming language*. http://chapel.cray.com/.

[4] Numrich, R. W. and Reid, J. 1998. *Co-array Fortran for parallel programming*. SIGPLAN Fortran Forum 17, 2 (Aug. 1998).

[5] *Titanium: A high-performance Java dialect*. http://titanium.cs.berkeley.edu/.

[6] *SHMEM API for parallel programming*. http://www.shmem.org/.

[7] Quadrics Ltd. *Quadrics SHMEM programming manual*. 2006.

[8] Cray, Inc. *Man Page Collection: Shared Memory Access (SHMEM)*. (S-2383-23).

[9] Parzyszek, K. 2003 *Generalized Portable Shmem Library for High Performance Computing*.Doctoral Thesis. UMI Order Number: AAI3105098., Iowa State University.

[10] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. Panda. *High Performance Remote Memory Access Comunications: The ARMCI Approach*. International Journal of High Performance Computing and Applications, Vol 20(2), 233-253p, 2006.

[11] Dan Bonachea. *GASNet Specification v1.1*. UC Berkeley Computer Science Division Report CSD-02- 1207, 2002.

[12] *MPI Standard*. http://www.mcs.anl.gov/research/projects/mpi/.

[13] *The OpenMP API specification for parallel programming*. http://openmp.org/wp/.

[14] *GASNet communication layer*. http://gasnet.cs.berkeley.edu/

[15] Nishtala, P. Hargrove, D. Bonachea, K. Yelick.*Scaling Communication-Intensive Applications on BlueGene/P Using One-Sided Communication and Overlap*, 23rd International Parallel & Distributed Processing Symposium (IPDPS), 2009.

[16] T. von Eicken: Active Messages: *An Efficient Communication Architecture for Multiprocessors*, Ph.D. thesis at University of California at Berkeley(1993).

[17] W.Jiang, J. Liu, H.Jin, D.Panda, W.Groop, R.Thakur *High Performance MPI-2 One-Sided Communication over InfiniBand.* In Proceedings of the 4th IEEE/ACM Int'l Symposium on Cluster Computing and the Grid (CCGrid 2004), April 2004.

[18] C. Bell and R. Nishtala *Firehose: An Algorithm for Distributed Page Registration On Clusters of SMPs*, May 2004. http://gasnet.cs.berkeley.edu/bell-nishtala–firehose-smp.pdf.

[19] *GASNet inter-Process SHared Memory (PSHM) design*. http://gasnet.cs.berkeley.edu/dist/docs/pshm-design.txt

[20] B. W. Kernighan and D. M. Ritchie. *The C Programming Language (Second Edition)*. Prentice Hall.

[21] *Berkeley UPC - Unified Parallel C*. http://upc.lbl.gov/

[22] Huang, J., Kumar, S. R., Mitra, M., Zhu, W., and Zabih, R. 1997. *Image Indexing Using Color Correlograms*. In Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97) (June 17 - 19, 1997).CVPR.IEEE Computer Society, Washington, DC, 762.

[23] H. Su, M. Billingsley, and A. George, *Parallel Performance Wizard: A Performance Analysis Tool for Partitioned Global-Address-Space Programming*, 9th IEEE International Workshop on Parallel & Distributed Scientific and Engineering Computing (PDSEC) of IPDPS 2008, Miami, FL, Apr. 14-15, 2008.